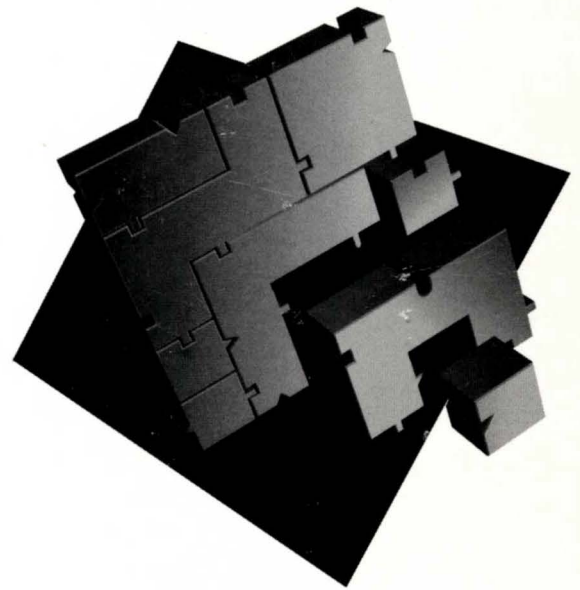




A
V
E
R
S
I
O
N
F
O
R
A
I
X
®

TALIGENT
TOOLS
FOR AIX®



TOOLS
TESTING ENVIRONMENT
SNIFF+™ DOCUMENTATION

TALIGENT TOOLS FOR AIX

TALIGENT, INC.
10201 NORTH DE ANZA BOULEVARD
CUPERTINO, CALIFORNIA 95014-2233
USA
(408) 255-2525

TALIGENT TOOLS FOR AIX

Copyright © 1994 Taligent, Inc. All rights reserved.
10201 N. De Anza Blvd., Cupertino, California 95014-2233 U.S.A.
Printed in the United States of America.

This manual and the software described in it are copyrighted.
Under the copyright laws, this manual or the software may not be copied, in whole or part, without prior written consent of Taligent. This manual and the software described in it are provided under the terms of a license between Taligent and the recipient and its use is subject to the terms of that license.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. and International Patents.

TRADEMARKS: Taligent and the Taligent logo are registered trademarks of Taligent, Inc. All other trademarks belong to their respective owners.

CONTENTS

Preface	XIII
<hr/>	
Part 1 Tools	2
Chapter 1	
Introduction	5
Chapter 2	
The build environment	7
Taligent build terminology	7
The build process	8
Makefiles	9
Makefile description syntax	9
Target types	9
Makeit	10
Passing options to make	11
Creating makefiles	11
Universal.Make	11
Environment variables	12
Setenv	13
How to change environment variables	13
When to change environment variables	13
Real life examples	14
A simple sample	14
A faster build	16
A clean build	17
A not-so-simple makefile	17
A simple *.PinkMake	18
Adding link libraries	19

Chapter 3	
Taligent build tools	23
CreateMake	24
FindSymbols	25
IPCurge	27
MakeExportList	27
Makeit	28
MakeSharedApp	30
MakeSharedLib	30
MakeSOL	31
mop	31
RunDocument	32
SharedLibCache	33
slibclean	33
SmartCopy	34
StartPink	34
StopPink	35

Chapter 4	
CreateMake	37
application	38
binariessubfolderdir	38
binary	39
build	40
compileoption	40
developmentobject	41
end	41
export	42
header	43
headerdir	43
heapsize	44
library	44
link	44
loaddump	45
local	46
localheader	46
localheaderdir	46
make	47
object (tag)	47
object (target)	48
objectdir	48

parentobject	49
parentobjectdir	49
private	50
privateheaderdir	50
program	51
public	51
server	52
source	52
sourcedir	53
start	53
subfolder	54
subfolderdir	54
subproject	55
testapplication	55
testlibrary	55
testparentobject	56
testserver	56
tool	56
trimdependencies	57

Chapter 5

Analysis tools	59
Overview	60
Tools	61
Limitations	61
TLocalHeapMonitor	61
TLocalHeapAnalyzer	62
Heap monitoring file format	62
Heap analysis file format	63
Heap corruption	64
Debugging heap corruption	64
AIX notes	64
Dynamic analysis	65
Dynamic typing	65
Dynamic error detection	65

Chapter 6	
Xcdb	67
Setup	69
Installation	69
Signals	70
Compiling	70
Running	70
Program starting	72
Program interrupting	72
Program terminating	72
Xcdb exit codes	72
Window organization	73
Window manipulation	74
Execution control	75
Format Control	76
Common Formats	77
Type-specific Formats	77
class, struct, and union formatting	79
Array formatting	80
Pointer formatting	83
Breakpoints	83
Preferences	84
Self-displaying C++ objects	85
Customization	86
Frequently asked questions	88
Reporting bugs	93
Appendix A	
Tips & techniques	95
cdpath	95
xcdb—the debugger	96
OpusBug()	96

Part 2 Test Environment	98
Chapter 7	
Test Environment Overview	101
Chapter 8	
Test framework	103
Test framework overview	104
TTest Class	104
Related classes	106
Getting started with the test framework	106
Designing a test	107
Creating a test	108
Writing a test function	109
Setting up the environment	110
Cleaning up after a test	110
Writing a test to run more than once	110
Overriding inherited MCollectible members of TTest	110
Writing text to the console	111
Combining tests	112
Using a script to run multiple tests	112
Combining operations in a single test class	112
Combining multiple TTest objects into a single test	114
Creating tests with dependencies on other tests	114
Identifying what a test does	115
Chapter 9	
RunTest	117
Performing a test	118
Testing an interface inherited from a base class	118
Providing input for a test	119
Parsing text inputs to a test	119
RunTest options	121
Stopping a test	122
Examining test results	123
Collecting timing information	123
Handling exceptions	123

Part 3 SNIFF+ Guide	124
Chapter 10	
Getting started	127
Introduction	127
About SNIFF+ documentation	128
Terminology	129
Typographical conventions	129
Basic SNIFF+ concepts	130
Main tools	130
Information extraction	130
Updating information	131
Project concept	131
Browsers and editors	131
Shortcuts	132
Prerequisites	133
Installation	133
Checking the environment	133
Copying the example files to your local directory	134
Starting the SNIFF+ tool	135
SNIFF+ command line	135
Starting SNIFF+ from shell	135
Creating a new project	136
Creating the filebrowser project	136
Setting the project attributes	138
Checking the source files into the version control system	140
Loading a subproject	142
Examining the results	143
Saving the new project and closing the Project Editor	143

Chapter 11	
Using SNIFF+	145
Browsing Symbols	145
Opening a	
Symbol Browser	145
Project tree	147
Constraining the	
list with filters	147
Type pop-up menu	148
Top-down browsing	148
Viewing JButton in the class hierarchy	148
Browsing the elements of JButton—the Class Browser	150
Studying protocols	152
Bottom-up browsing	154
Studying the method GetMinSize	154
Where glook is used—the Retriever	156
Retrieving session 2—getting information about menu handling	158
Editing	158
Loading a symbol into the editor	159
Working with the Symbol list	160
Checking out a file	160
Some useful editing helps	161
Viewing and editing class and member descriptions	162
Opening the Documentation Browser	162
Viewing other descriptions	163
Changing from read-only to writable	163
Editing the file	164
Checking in and checking out files	164
Adding Taligent public includes to a new project	164
Compiling	165
Starting the compiler	166

Chapter 12	
Basic Elements	169
SNIFF+ architecture	169
SNIFF+ environment	169
Basic user-interface components	170
Status line	170
Layout handle	170
Common dialogs and windows	171
Find Dialog	171
File Dialog	172
Directory Dialog	174
Print Dialog	175
About dialog	175
License dialog	176
Progress Window	176
Error log window	177
Common menus	177
Icon menu	177
Info menu	179
Class menu	180
Filter menu	181
History menu	181
Edit pop-up menu	182
Shortcuts	183
Keyboard shortcuts	183
Chapter 13	
SNIFF+ subsystems	185
Workspace manager	185
Project menu	186
Project Editor	187
File list	188
Project tree	188
File menu	188
Make menu	189
Project menu	190
Show Locking button	191
Project Editor with locking information shown	192
Source Files dialog	197
Project Attributes Dialog	199
Project Attributes dialog for frozen subprojects	203
Symbol browser	204
Type pop-up	205
Project tree	205

Class browser	206
Inheritance Graph	207
Hide Overridden button	207
Type pop-up	207
Hierarchy browser	208
Hierarchy menu	209
Projects menu	209
Retriever	210
Project tree	211
Status line	211
Retrieve menu	211
Filter menu	212
Editor	213
Symbol List	214
Class pop-up	214
File menu	214
Edit menu	216
Positioning menu	217
Utilities menu	218
Make menu	218
Exec menu	219
Inspect menu	219
Build menu	220
TAE menu	220
Custom menus	221
Debugging mode	222
Editing shortcuts and goodies	223
Documentation Browser	224
Symbol List	225
Class pop-up	225
File menu	225
Edit menu	226
Styles menu	226
Info menu	226
Class menu	226
TAE menu	226
Custom menus	226
Shell	227
Edit menu	227
Shell menu	228
Target menu	228

Chapter 14

Customizing your environment	231
Preferences	231
Site	231
Preferences dialog	232
Teamwork Support	234
Overlaying shared files	234
Information extractor (sniffserver)	235
Running the sniffserver on a different host	235
Dealing with preprocessor macros	236
Configuring the parser	237
Files created and used by SNIFF+	239
Project file	239
ETRC file	239
Parser configuration file	239
Retriever filters file	240
Template files	240
Custom menu file	240
Error formats file	241
Files generated by SNIFF+ and stored in the generate directory	242
Tuning and persistency of symbolic information	243
File-level symbol persistency (default)	243
Project-level symbol persistency	243
Comparison of project loading times	244
SNIFF+ projects	244
Projects in SNIFF+	244
Declaration and implementation files in separate directories	245
Projects with many subprojects	246

Chapter 15

Support for other functions	249
Makefile Support	249
Dependencies (dependencies.incl)	249
Object file list (ofiles.incl)	250
Emacs integration	250
Integrating Emacs	251
Working with Emacs and SNIFF+	252
Configuring the Emacs integration	253
How the Emacs integration works	254
Version control	254
Restrictions in using RCS and SCCS with SNIFF+	255
Working with SNIFF+ version control and locking	255
How RCS and SCCS are integrated	255
SNIFF+ locking	255

Appendix B

GNU Regular Expressions 257

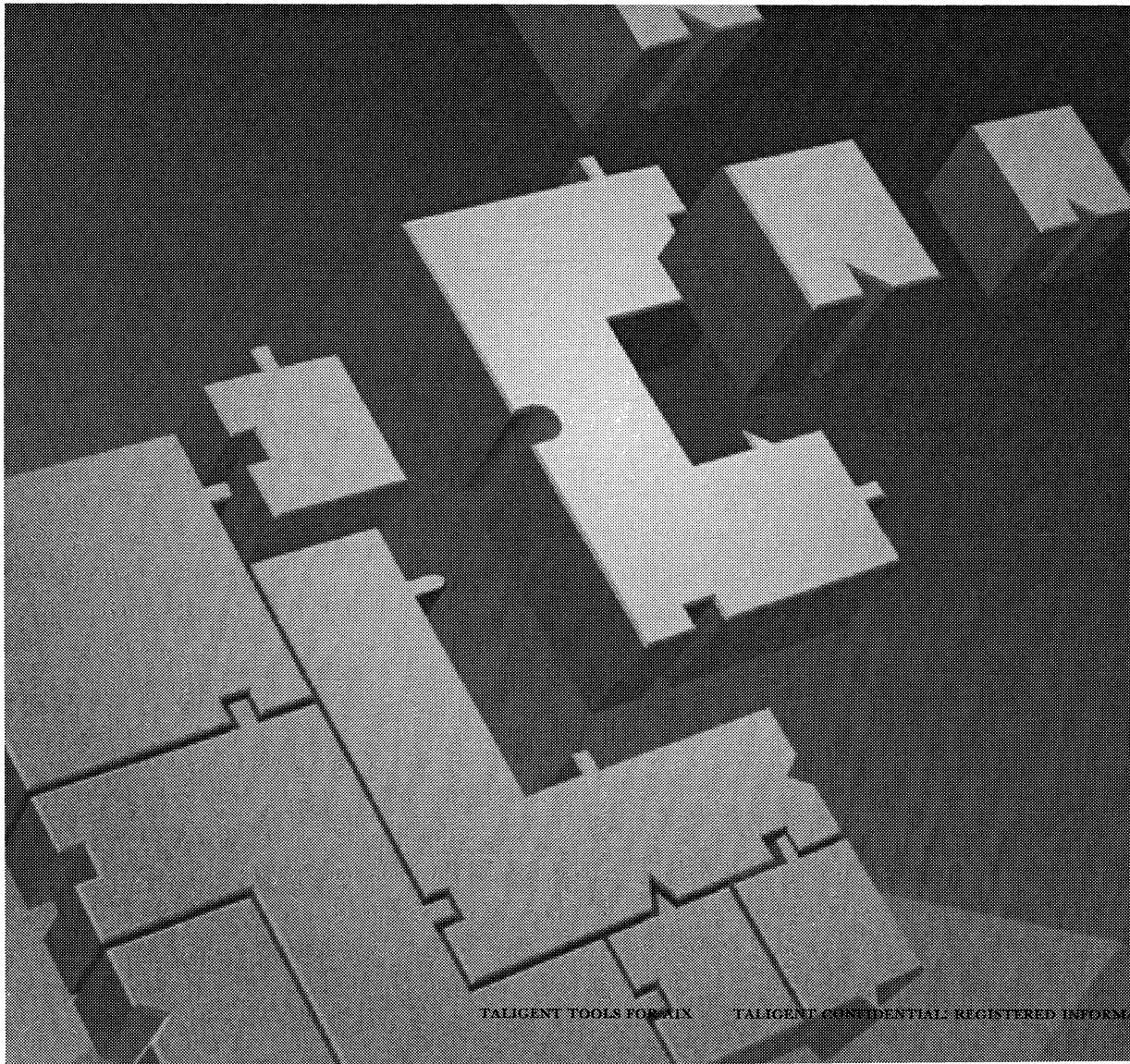
 Syntax 257

Appendix C

ETRC file entries 261

 Description of entries 261

Index 269



PREFACE

Taligent Tools for AIX is a reference guide to the tools that Taligent engineers use in everyday development work on the AIX[®] platform. Most of these tools were developed specifically for building the Taligent Application Environment[®].

This guide has three parts:

Taligent Tools describes the tools in detail, and provides information about how to use them both collectively and individually.

Test Frameworks covers the test frameworks that you use to test your programs.

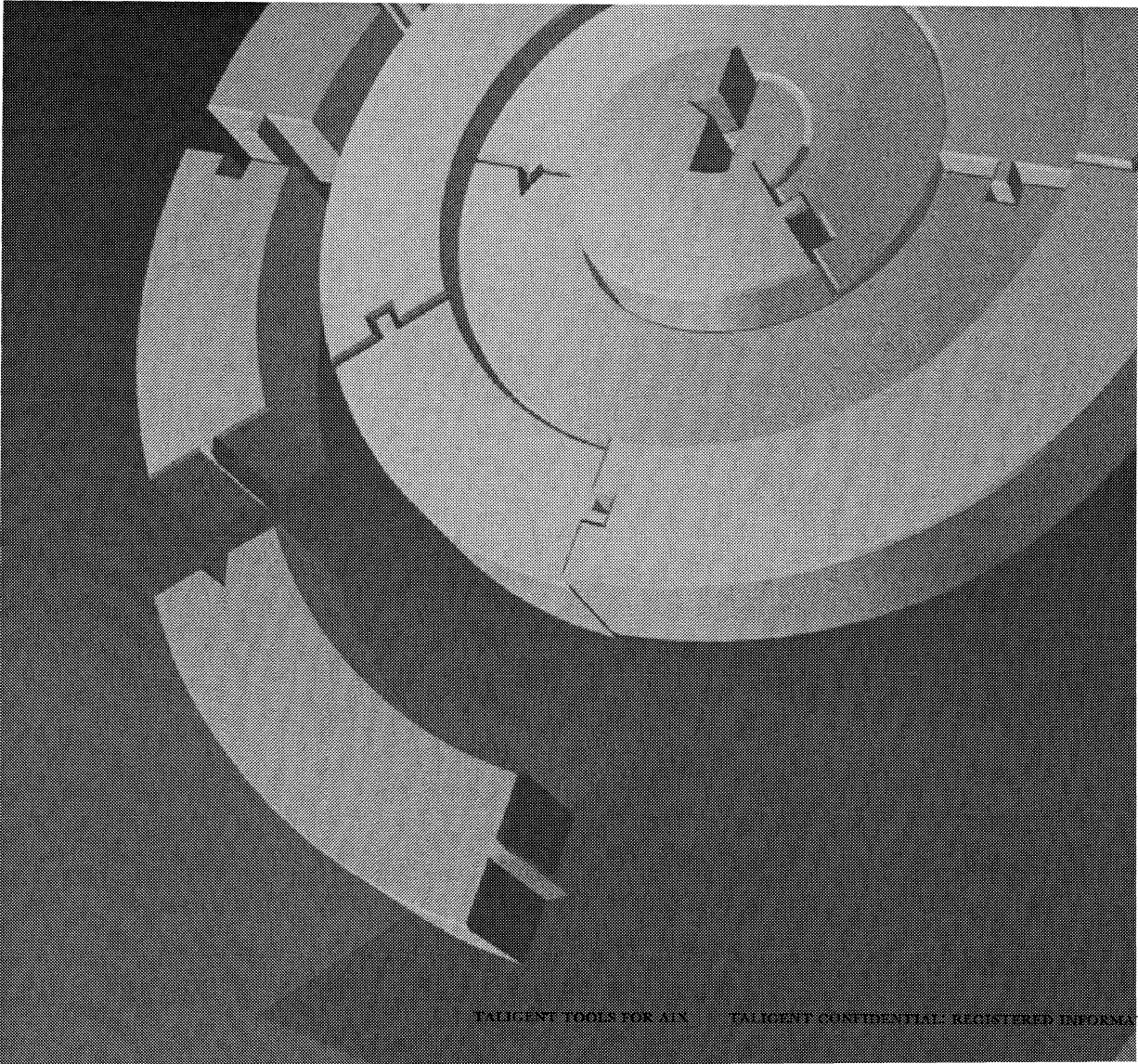
SNIFF+ Guide is a user's guide to the SNIFF+ development environment.

PART 2

TOOLS

Chapter 1 Introduction	5
Chapter 2 The build environment	7
Chapter 3 Taligent build tools	23
Chapter 4 CreateMake	37
Chapter 5 Analysis tools	59
Chapter 6 Xcdb	67
Appendix A Tips & techniques	95

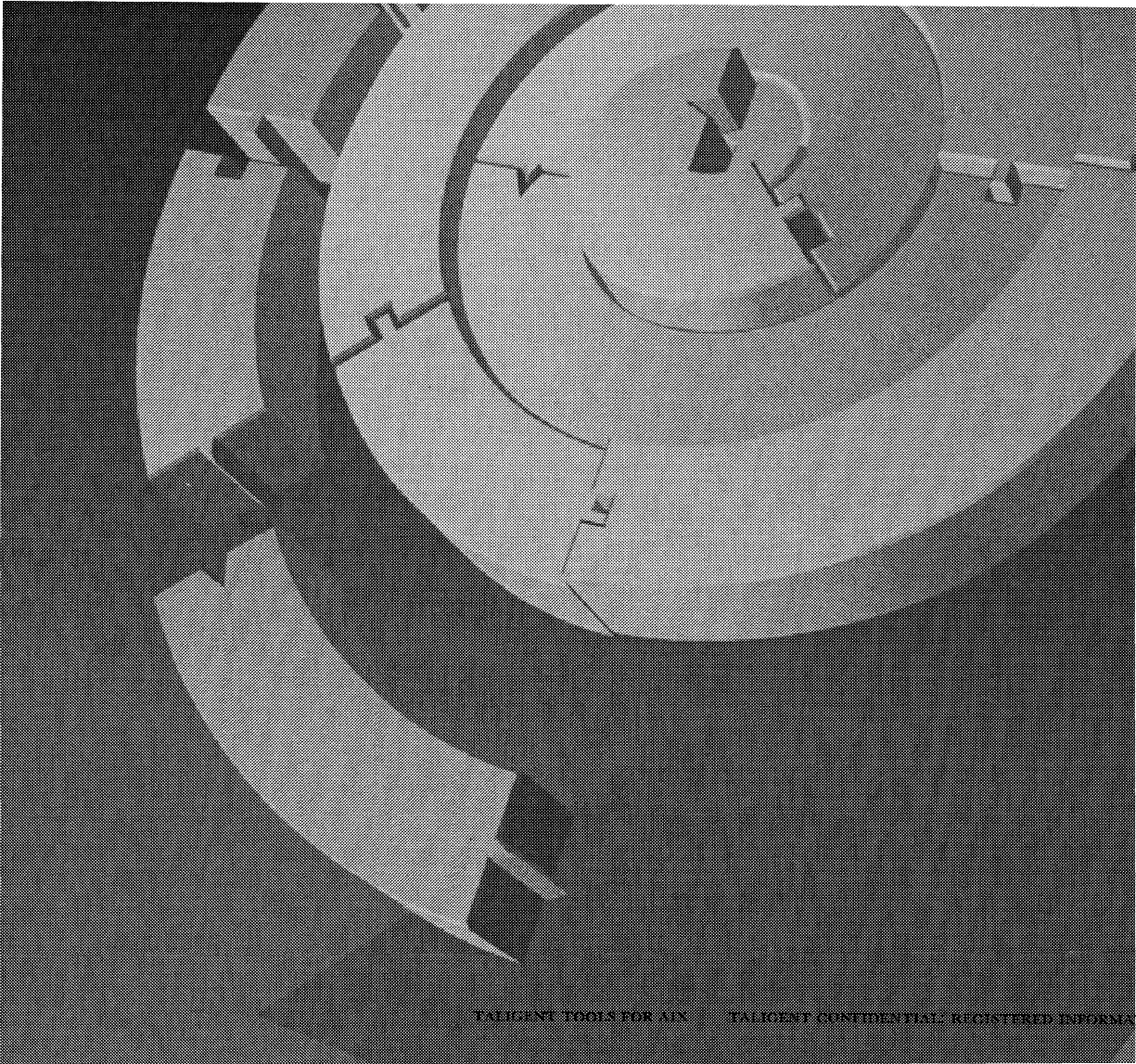
z



CHAPTER 1

INTRODUCTION

Taligent Tools for AIX describes the Taligent development tools and how to use them. Advanced Interactive Executive (AIX) This guide assumes that you are running the C Shell (csh) which is the standard shell used for the Taligent build environment. If you intend to use a different UNIX Shell, refer to the documentation appropriate for that shell.



THE BUILD ENVIRONMENT

The Taligent AIX build environment was designed to allow individual contributors to efficiently accomplish their work, to allow full-system (or major subsystem) builds—and to accomplish both in a similar fashion. Once you know how to do the first, the second is easy. This chapter focuses on how you, the individual contributor, use the build environment.

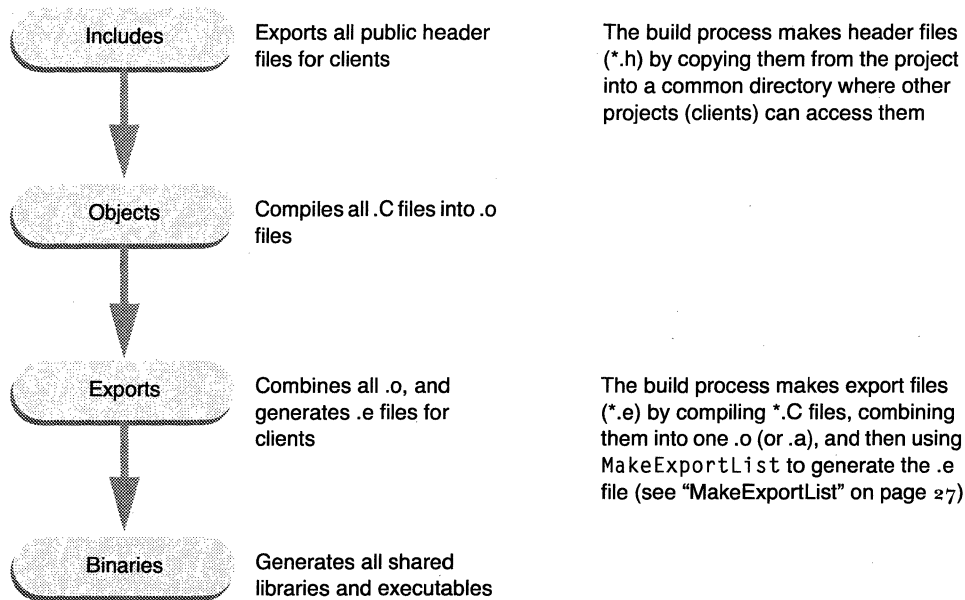
TALIGENT BUILD TERMINOLOGY

Taligent uses these terms when describing the build environment:

- ※ *Build*—run the necessary tools to generate client and executable files in the proper order on any project or any project hierarchy. To accomplish this, each project (or project hierarchy) must have its own makefile. See “Makefiles” on page 9 for more information.
- ※ *Client files*—headers and export files.
- ※ *Header files* (.h files)—files containing your C++ class definitions.
- ※ *Export files* (.e files)—files containing a list of all entry points in your shared library. Your clients link against .e files and the runtime system binds the calls to your shared library at run time.
- ※ *Binaries*—executable programs or applications that use shared libraries during execution.
- ※ *Shared libraries*—Class libraries used by multiple programs are usually loaded dynamically at runtime. To build a shared library, compile your source files, generate your .e file, and link against other .e files. For building applications, use `MakeSharedLib` (see page 30 for more details).
- ※ *Executables*—binaries or shared libraries. To build a program or executable, compile your source files and link against .e files using `MakeSharedApp`. Your source files must contain a main entry point. (See “`MakeSharedApp`” on page 30 for more details.)

THE BUILD PROCESS

The Taligent Application Environment is a big web of interdependencies. To solve these interdependencies, the build process occurs in four phases that first build all client files, and then build all executables. This automated process generates both client and executable files.



NOTE For Taligent Operating System builds, files currently have different extensions than those cited in the illustration: object files are *.ip, libraries are *.lib, and export files are *.client.ip.

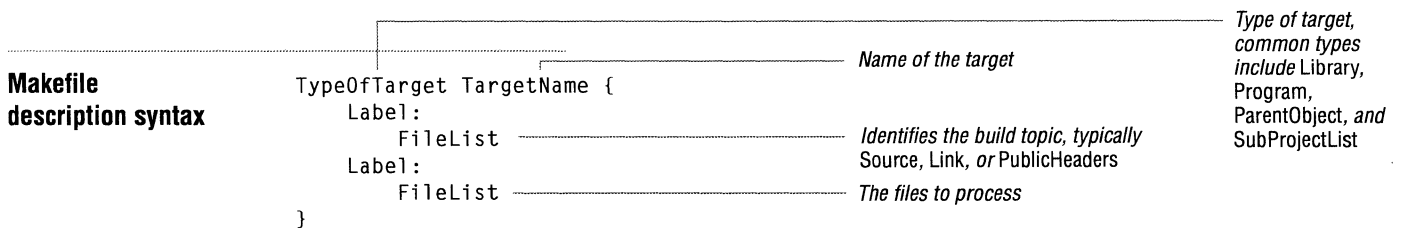
To automate the build process, use makefile descriptions to specify the files to build, and use CreateMake to translate the makefile descriptions and to build the files.



CAUTION The current build tools do not test to see if your component, application, or library has the same name as one used by the system. The build process will automatically overwrite the Taligent file with yours if you have a duplicate name.

MAKEFILES

The makefiles associated with each project are *makefile descriptions*, not standard makefiles. During a build, Makeit calls CreateMake to translate the makefile description to a standard-makefile. Makeit then calls make to analyze the dependencies of the generated makefiles and update the project. Because makefile descriptions are source code, you can check them in to SCM; but, do not check in the generated makefiles. Makefile descriptions have filenames in the form *Project.PinkMake*, where *Project* is the name of the project or directory.

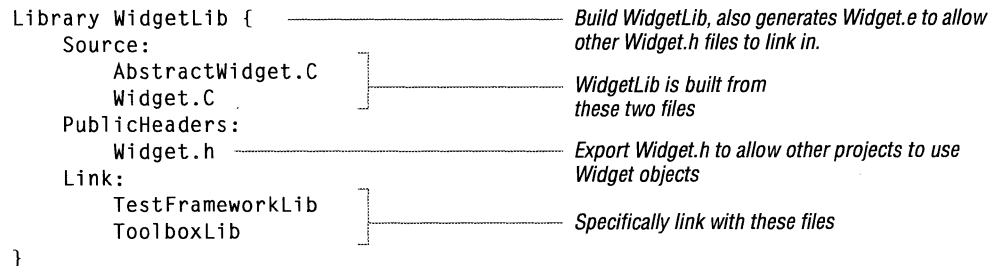


Target types

CreateMake generates different build rules for each type of target. Here are a few common target descriptions

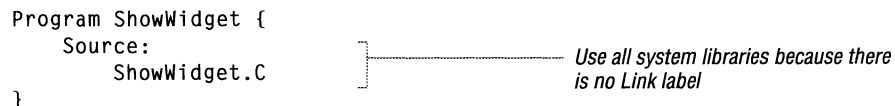
Library

Generates rules to build a shared library.



Program

Generates rules to build an application.



ParentObject

Generates rules to build and combine the source files. Frequently used to combine several projects into one larger library.

```
ParentObject FooBarLib {
  source:
    Foo.C
    Bar.C
  publicheaders:
    Foo.h
    Bar.h
}
```

Generate FooBarLib.o to be included in the build of another library

ParentObject targets do not require a Link label because they are not linked

Exported for clients

SubProjectList

A special type of target that lists all the sub projects that you want to build; it does not have a target name or any labels. Makeit uses this list when traversing the project hierarchy and only builds from those directories listed.

```
SubProjectList {
  SubProj1
  SubProj2
}
```

Build SubProject1 and SubProject2, but ignore SubProject3, even though it is part of the project

MAKEIT

Once you have a makefile description, use Makeit to build your project. Makeit is a specialized wrapper (or front end) to make. Makeit simplifies builds, provides consistency, and has the ability to traverse project hierarchies and convert makefile descriptions to real makefiles along the way.

Syntax

Makeit [options] [Targets]

Makeit only has a few options. If you specify any other options, Makeit passes them along to make. So in effect, Makeit has the same options as make. For information about Makeit and its options, see “Makeit” on page 28.

If you omit options and targets, Makeit goes through each target in the build process (Includes, Objects, Exports, and Binaries), and builds the necessary dependencies. However, because Makeit is really a wrapper for make, it accepts any legitimate target in a makefile.

Makeit DemoApp

A common mistake is to build one target (like the previous example), and not realize that Makeit is going to do a make on all subprojects of DemoApp—many of which do not have a target DemoApp. To prevent Makeit from building subprojects, include -c.

Makeit -c DemoApp

For more robust examples, see “Real life examples” on page 14.

Passing options to make

Makeit passes any options it does not recognize. You can use this feature to pass options to make. Makeit passes arguments to options, and can override variables in makefiles. For example, to override the COPTS variable in the makefile:

```
Makeit COPTS=-g Binaries
```

Creating makefiles

When Makeit builds a makefile on the fly, it does so because either

- ⌘ The *.Make file does not exist
- ⌘ The *.PinkMake file is newer than the *.Make file, or
- ⌘ The -M option forced automatic makefile generation.

Makeit uses CreateMake to translate the makefile descriptions (*.PinkMake) to UNIX makefiles (*.Make).

Universal.Make

To prevent duplication in each makefile, and to allow more flexibility, CreateMake includes Universal.Make in every generated makefile (*.Make).

Universal.Make contains global targets and rules. Some of the familiar global targets are: Includes, Objects, Exports, and Binaries. Other targets are useful because they are applied only to the projects in the build and not to every directory in the hierarchy. For example you can have a subsystem that is checked into SCM, but is not part of the build. These targets will not be applied to those projects.

Other Global Targets

Global Target	Task
Clean	Remove all .o's, .e's, and libraries that were built.
Complete	Expand into the standard targets: Includes, Objects, Exports, and Binaries.
Makefiles	Allows you to traverse the directory and rebuild makefiles as needed.

The includes, objects, exports, binaries, and clean targets have lower-case synonyms, so capitalization is not required.

ENVIRONMENT VARIABLES

The AIX build environment relies heavily on two types of environment variables:

Pathname environment variables contain pathnames that are specific to each user. All the build tools and makefiles refer to the standard locations through environment variables. This allows you to define the location of your working directories.

`$TaligentRoot`, set by `Setenv`, is the basis for all other pathname variables. For example, here are two pathnames as set by `Setenv`:


The {} bound variable references in shell scripts. ————— `setenv TaligentIncludes ${TaligentRoot}/PinkIncludes`
`setenv TaligentExports ${TaligentRoot}/Exports`

Variable	Path to
<code>LIBPATH</code>	Taligent shared libraries used during runtime.
<code>TaligentBinaries</code>	Taligent runtime binaries.
<code>TaligentDefaultHomePlace</code>	Repository for the current user's home place (Only one user currently for the system.) The Workspace group will provide a better object API for getting access to the current user and storage areas related to that user in future releases.
<code>TaligentExports</code>	Taligent shared library interface files that developers link with to access Taligent shared libraries.
<code>TaligentExtensionIncludes</code>	Directory containing interfaces to system extension developers.
<code>TaligentIncludes</code>	Main <code>#includes</code> directories used in Taligent builds.
<code>TaligentIncludesDir</code>	Base parent directory of all Taligent <code>#includes</code> (this is the parent of <code>\$TaligentIncludes</code> , <code>\$TaligentExtensionIncludes</code> , and <code>\$TaligentObsoleteIncludes</code>).
<code>TaligentLibs</code>	Directory for certain nonshared libraries.
<code>TaligentPlacesRoot</code>	Repository where Places for the machine reside.
<code>TaligentRoot</code>	The base of everything in the build and runtime system.
<code>TaligentSharedLibs</code>	Taligent runtime shared libraries.
<code>TaligentSystemDataRoot</code>	Repository for system data files. These are typically configuration files, not first class user data such as movies, images, or sounds.
<code>TaligentTemporaries</code>	Repository for temporary files until people use real Pluto temporary file support.
<code>TaligentUniversalMake</code>	Universal.Make file used in build system.

Option environment variables contain the standard options to the standard tools that the build uses. Having the options in an environment variable allows you to change and experiment with certain options (like debugging options) without disturbing others. Never add options to the compiler (or to any build tools) in the makefile—use the environment variables instead.

Makefile variables are a common alternative to environment variables, but are disastrous in our build environment

Variable	Options to
CompileOptions	x1C command line during builds as the options for building Taligent code and default search paths to Taligent #includes.
MakeSharedAppOptions	MakeSharedApp as default options for building a Taligent shared library.
LinkOptions	x1C link command line during builds.

 **NOTE** Occasionally a project requires a special option (such as working around a compiler bug). For special cases when the project cannot build or will not work unless it has a particular option, add the option to the makefile description file (*.PinkMake). To add an compiler option, add the following line to the *.PinkMake file:

```
compileoptions: -NewOptions
```

Setenv

Setenv defines the standard values for all the environment variables that the Taligent build environment requires. Always use Setenv to set the variables and pathnames. If you need to change a variable, do so after running Setenv.

How to change environment variables

The easiest way to change an environment variable is to add to it. For example, in a shell script, to add `-D__MYDEBUG__` as an option to the compiler:

```
setenv CompileOptions "-D__MYDEBUG__ ${CompileOptions}"
```

If you frequently add the same option, put the setting in a startup file.

When to change environment variables

It is easy to change the environment variables to customize your environment, but be careful not to get too carried away with additions. Remember, other people need to build your project too; do not become dependent on a particular `-D` you have defined in your environment variable. The system builds use the default options as defined in the BuildOptions file.

REAL LIFE EXAMPLES

By now you should understand the organization of projects and have a fundamental grasp of how the build works. This section ties together everything you have learned by using several real life examples.

A simple sample

SimpleSample is similar to Kernighan and Ritchie's *hello world* program. This program is ideal for demonstrating how to create, build, and execute an application.

How to create SimpleSample

- 1 Create a directory named `SimpleSample`. You can create the directory anywhere on your file system; in your home directory is probably best.

- 2 Create a source file `hello.C` and enter:

```
#include <stdio.h>
void main()
{
    printf("Hi there everybody!\n");
}
```

Use your favorite editor to create `hello.C`. For custom features that can improve Emacs efficiency, see “” on page 97.

- 3 Create a makefile description called `SimpleSample.PinkMake` and enter:

```
program SimpleSample {
    source:
        hello.C           // A single source file
}
```

The name of the `*.PinkMake` file must be the same as the name of the directory in which it resides. The example resides in `.../SimpleSample`.

- 4 Build `SimpleSample` using `Makeit` without any options or targets (See the section `Makeit`, “Default operation:” on page 22):

```
Makeit
```

The build log	What follows is the build log; yours should look similar.
Makeit messages	<p>The first message is from Makeit stating that it did not find SimpleSample.Make in the project. Therefore, Makeit built a makefile from SimpleSample.PinkMake. Line 3 is the CreateMake command that Makeit issued to create the makefile.</p> <pre>1 ### Makeit: No makefile found in `/home/EeeDee/SimpleSample'. 2 ### However one will be built from `SimpleSample.PinkMake'. 3 # CreateMake > SimpleSample.Make;</pre>
The Includes phase	<p>Since SimpleSample.PinkMake did not specify any public header files, Makeit did not build any include files.</p> <pre>4 # 5 # Making "Includes" for "/home/EeeDee/SimpleSample"... 6 # make -f SimpleSample.Make Includes 7 # 8 make: Nothing to be done for 'Includes'.</pre>
The Objects phase	<p>Compiles hello.C to hello.o, and contains the make line that Makeit called.</p> <pre>9 # 10 # Making "Objects" for "/home/EeeDee/SimpleSample"... 11 # make -f SimpleSample.Make Objects 12 # 13 # Compile hello.C to produce hello.o</pre>
The Exports phase	<p>Did not build a shared library because SimpleSample did not build an export file.</p> <pre>14 # 15 # Making "Exports" for "/home/EeeDee/SimpleSample"... 16 # make -f SimpleSample.Make Exports 17 # 18 make: `Exports' is up to date.</pre>
The Binaries Phase	<p>Creates the executable application by calling MakeSharedApp (as echoed from make). For more information, see “MakeSharedApp” on page 41</p> <pre>19 # 20 # Making "Binaries" for "/home/EeeDee/SimpleSample"... 21 # make -f SimpleSample.Make Binaries 22 # 23 MakeSharedApp -L. -L/usr/lib/dce -o SimpleSample hello.o /home/EeeDee/work/Exports/RuntimeLib.e /home/EeeDee/work/Exports/OpixLib.e /home/EeeDee/work/Exports/ToolboxLib.e /home/EeeDee/work/Exports/TimeLib.e /home/EeeDee/work/Exports/TestFrameworkLib.e /home/EeeDee/work/Exports/HighLevelAlbert.e /home/EeeDee/work/Exports/LowLevelAlbert.e /home/EeeDee/work/Exports/AlbertPixelBuffers.e</pre>
The Copy phase	<p>Copies the built application to \$TaligentBinaries, the standard location for executable files, and leaves a copy in the current directory.</p> <pre>24 SmartCopy SimpleSample /home/EeeDee/work/TaligentBinaries</pre>

**How to execute
SimpleSample**

When the build completes, execute SimpleSample program by typing its name at the UNIX prompt. It should look like this:

```
% SimpleSample
OPIX compile timestamp = Jan 22 1994, 08:25:22  ----- The Taligent AIX Layer prints a time-stamp
Hi there everybody!                               when it runs an application.
%
```

A faster build

A slightly faster and more efficient way to use Makeit is to include the target name. For example, change SimpleSample to use a Taligent object, and then rebuild it.

- 1 Change hello.C to look like this:

```
#include <Geometry.h>

void main()
{
    TGRect unusedRect(0, 1, 2, 4);
    unusedRect.PrintObject();    // Print coordinates
}
```

- 2 Rebuild the application.

```
Makeit SimpleSample.
```

The build log looks similar to this:

```
#
# Making "SimpleSample" for "/home/EeeDee/SimpleSample"...
# make -f SimpleSample.Make SimpleSample
#
# Compile hello.C to produce hello.o
MakeSharedApp -L. -L/usr/lib/dce -o SimpleSample hello.o /home/EeeDee/work/Exports/RuntimeLib.e /home/EeeDee/work/Exports/OpixLib.e /home/EeeDee/work/Exports/ToolboxLib.e /home/EeeDee/work/Exports/TimeLib.e /home/EeeDee/work/Exports/TestFrameworkLib.e /home/EeeDee/work/Exports/HighLevelAlbert.e /home/EeeDee/work/Exports/LowLevelAlbert.e /home/EeeDee/work/Exports/AlbertPixelBuffers.e
```

Running the new SimpleSample should print these results:

```
%SimpleSample
OPIX compile timestamp = Jan 22 1994, 08:25:22
TGRect (top = 1.000000, left = 0.000000, bottom = 4.000000, right = 2.000000)
%
```

A clean build

To ensure a successful build, delete all the object files before you build a project (or project hierarchy). Clean instructs Makeit to delete the object files before building the project.

```
Makeit Clean Complete
```

A not-so-simple makefile

TuffyData is an application with several dependency files. This makefile description for TuffyData (TuffyData.PinkMake) is typical of a Taligent application.

```
// $Revision: 1.1 $
// Copyright (c) 1994 Taligent, Inc. All Rights Reserved.
```

Used by all compile commands. ————— `compileoption: -D__DEBUG__ -DUSE_FILE_SEGS`

Copy these make commands into the beginning of the generated makefile. —————

```
start {
TestHeaderDir=../../AES/UE/LocalIncludes

LocalIncludes ::
    test -d $(TestHeaderDir) || mkdir $(TestHeaderDir)
}
```

Directory of headers to export. ————— `localheaderdir: $(TestHeaderDir)`

Dependencies and makefile commands for creating the runtime library. —————

```
library CellModelLib {
publicheaders:
    CellModel.h
    CellModelView.h
    CellSelectionInteractor.h
source:
    CellModel.C
    CellModelView.C
    CellSelections.C
    CellModelCommands.C
    CellSelectionInteractor.C
link:
    GraphicDocumentLib
    StandardDocumentLib
    NewGraphicApplicationLib
    BDFTestLib
    CompoundDocumentLib
    BasicDocumentLib
    NewControlsLib
    ConstructorArchiveLib
    AlbertScreens
    {UniversalLinkList}
}
```

Create make dependencies for TuffyData, and build a single executable with these sources linked in.

```
binary CreateTuffyData {
source:
    CreateTuffyData.C
link:
    CellModelLib
    StandardDocumentLib
    GraphicDocumentLib
    NewGraphicApplicationLib
    BDFTestLib
    CompoundDocumentLib
    BasicDocumentLib
    NewControlsLib
    ConstructorArchiveLib
    AlbertScreens
    {UniversalLinkList}
}
```

A simple *.PinkMake

How do you determine which link files you need to specify in your *.PinkMake file? If you don't specify any link files, CreateMake links *all* library files. As you can imagine, this is not economical. Currently, the only way to determine which link files to include is by trial and error, and with a little help from FindSymbols.

Consider this makefile description called JustAView.PinkMake. JustAView builds a shared library and an application binary. To link all library files, create JustAView.PinkMake like this:

```
Shared library ----- library JustAViewLib {
                        source:
                            MyView.C
                        }

Main application binary ----- binary JustAView {
                                source:
                                    Main.C
                                }
```

Adding link libraries

To determine which library files to link, include `link: targets` and specify `{SimpleLinkList}` as the tag in each list. `{SimpleLinkList}` is a variable specifying a minimal set of libraries that most applications require:

```
library JustAVLib {
source:
    MyView.C

    link:
        {SimpleLinkList}      // Minimal set
    }

binary JustAView {
source:
    Main.C
```

Add link targets

```
    link:
        {SimpleLinkList}      // Minimal set
    }
```

Add link targets

```
    link:
        JustAVLib             // The JustAView library created above
        {SimpleLinkList}     // Minimal set
    }
```

When you build the `JustAView` project, `Makeit` will list errors for undefined symbols encountered when `MakeSharedLib` executes. In the messages, look for errors like these below the `MakeSharedLib` command line:

```
... MakeSharedLib -o JustAVLib ...
ld: 0711-317 ERROR: Undefined symbol: .TGArea::~TGArea()
ld: 0711-317 ERROR: Undefined symbol: .TRGBColor::~TRGBColor()
ld: 0711-317 ERROR: Undefined symbol: .TGRect::~TGRect()
ld: 0711-317 ERROR: Undefined symbol: __vtt12TContentView
```

To find the library files in which these symbols are defined, use `FindSymbols`. (The first time you run `FindSymbols`, it parses all library files and builds a database file so that subsequent lookups execute quickly.) To perform a lookup, run `FindSymbols` and specify the symbol exactly as it appears in the error listing. The symbol name must be enclosed within apostrophes (single quotes).

```
FindSymbols '.TGArea::~TGArea()'
```

Which produces a listing like this:

```
TGArea::~TGArea():
    HighLevelAlbert
```

This is the unique set of libraries identified:

Link tag to add ————— HighLevelAlbert

This listing indicates that the symbol is in `HighLevelAlbert`. Add that name as the tag in the library's `link: target`. To look for multiple symbols at once, include each as a separate argument on the `FindSymbols` command line:

```
FindSymbols '.TRGBColor::~TRGBColor()' '.TGRect::~TGRect()' '__vtt12TContentView'
```


Which produces this listing:

```
TRGBColor::~TRGBColor():
    LowLevelAlbert
TGRect::~TGRect():
    CommonAlbert
    HighLevelAlbert
    __vtt12TContentView:
    NewGraphicApplicationLib
```

This is the unique set of libraries identified:

```
CommonAlbert
HighLevelAlbert
LowLevelAlbert
NewGraphicApplicationLib
```

Notice that TGRect::~TGRect(): appears in CommonAlbert and HighLevelAlbert. When you get multiple libraries, you probably need to include only one. Try one and if you still get errors for the symbol, try the other. In a worst case, include both. This example only needed HighLevelAlbert.

```
library JustAViewLib {
    source:
        MyView.C

    link:
        HighLevelAlbert // Add
        LowLevelAlbert // Add
        NewGraphicApplicationLib // Add
        {SimpleLinkList}
}

binary JustAView {
    source:
        Main.C

    link:
        {SimpleLinkList}
}
```

Add link targets —————

Even if you lookup every symbol in the list, it probably won't be enough to build completely, because the libraries might also require other libraries. When you build JustAView again, you get these errors:

```
... MakeSharedLib ...
ld: 0711-317 ERROR: Undefined symbol: __vtt5TView
ld: 0711-317 ERROR: Undefined symbol: .TView::GetClassMetaInformation()
ld: 0711-317 ERROR: Undefined symbol: .TEventSenderSurrogate::GetClassMetaInformation()
```

Repeat the lookup and *.PinkMake modification until MakeSharedLib doesn't return an error.

Once your build gets past MakeSharedLib without error, you will probably find MakeSharedApp producing similar errors:

```
... MakeSharedLib ...
... MakeSharedApp ...
ld: 0711-317 ERROR: Undefined symbol: TView::virtual-fn-table-ptr-table
ld: 0711-317 ERROR: Undefined symbol: .TView::GetClassMetaInformation()
ld: 0711-317 ERROR: Undefined symbol: .TEventSenderSurrogate::GetClassMetaInformation()
ld: 0711-317 ERROR: Undefined symbol: .TInputDevice::GetClassMetaInformation()
ld: 0711-317 ERROR: Undefined symbol: .TViewRoot::~~TViewRoot()
ld: 0711-317 ERROR: Undefined symbol: .TViewRoot::TViewRoot(TRequestProcessor*)
ld: 0711-317 ERROR: Undefined symbol: .TViewRoot::AdoptChild(TView*)
```

Use FindSymbols again, but this time, add the link: tags to the binary target.

```
library JustAViewLib {
source:
    MyView.C

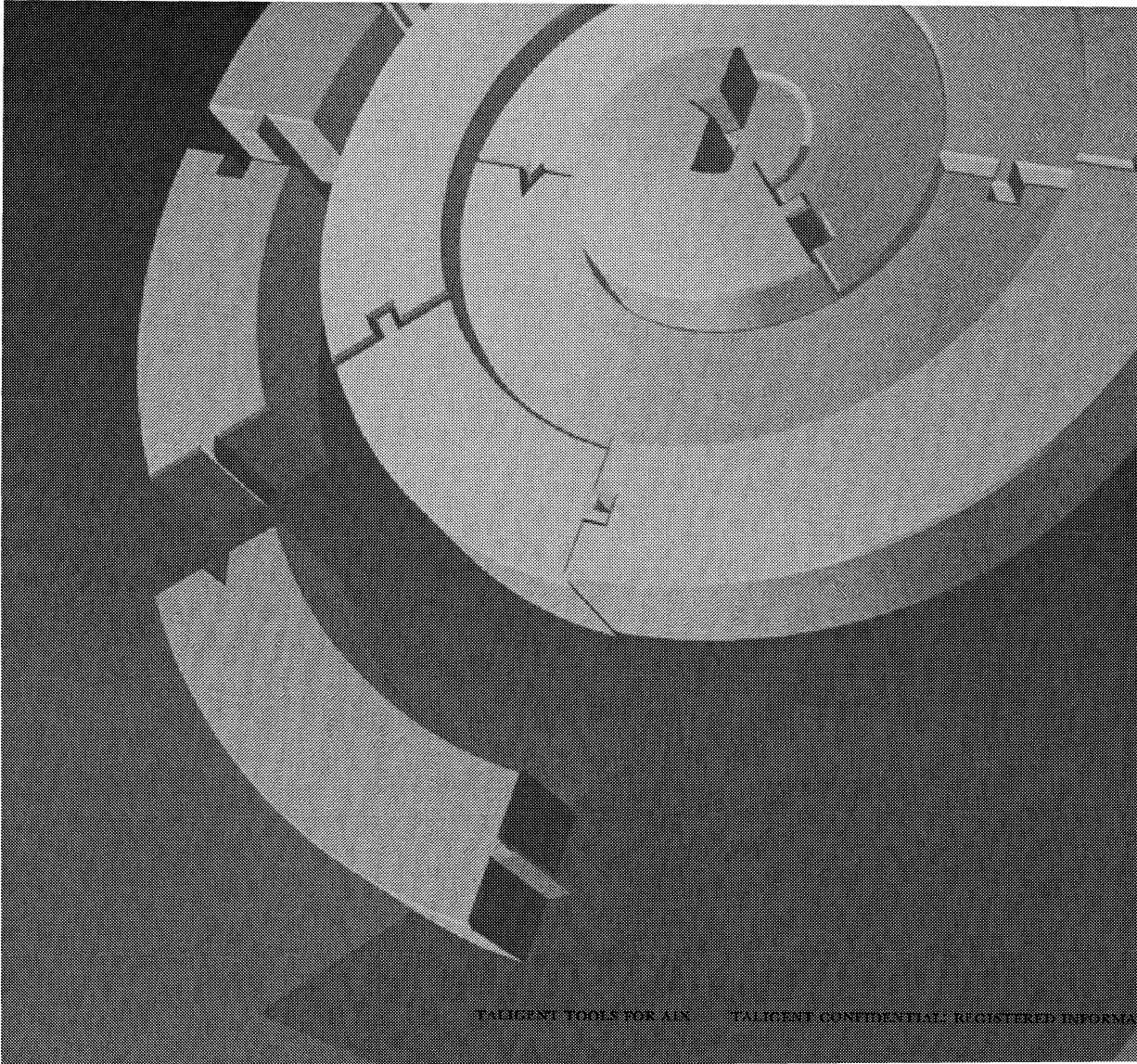
link:
    ViewSystemLib
    InputLib
    HighLevelAlbert
    LowLevelAlbert
    NewGraphicApplicationLib
    {SimpleLinkList}
}

binary JustAView {
source:
    Main.C

link:
    ViewSystemLib
    InputLib
    JustAViewLib
    {SimpleLinkList}
}
```

Add link targets

Repeat the process until Makeit completes the build.



CHAPTER 3

TALIGENT BUILD TOOLS

The Taligent build tools include tools and scripts that you run from the command line, and tools and scripts that those tools call. While this chapter documents how to run all of the Taligent build tools, there are some tools that you should avoid and are so noted. In addition, some tools require you to log on with super user access.

CREATEMAKE

CreateMake reads a file *Project.PinkMake* and creates a UNIX makefile for building the project. CreateMake writes the makefile to `stdout`; by convention, you should redirect the output to *Project.Make*.

Installation CreateMake is located in `/usr/taligent/bin` and requires no installation. Make sure this directory is in your command search path.

Syntax CreateMake [*sourcefile*] [-fast] [-D *define*]... [-I *includePath*]... [-nom] [-vers] > *outputfile*

Arguments	<code>-D <i>define</i></code>	Include the specified definition during processing.
	<code>-fast</code>	Preprocess the source files and create a single <code>.c</code> that <code>#includes</code> the source files to build each target. this results in faster builds, but is <i>not</i> to be used for final builds.
	<code>-I <i>includePath</i></code>	Add the path to the <code>#include</code> directory search-list.
	<code>-nom</code>	Generate a makefile that does not rely on Universal.Make for processing.
	<i>outputfile</i>	The file containing the new makefile. If you omit <i>outputfile</i> , output goes to <code>stdout</code> .
	<i>sourcefile</i>	The input file to process is usually a <code>*.PinkMake</code> filename. If you omit <i>sourcefile</i> , CreateMake assumes the current directory name is the project. For example, if the current directory is <code>/TestLib</code> , the <i>sourcefile</i> is <code>TestLib.PinkMake</code> .
	<code>-vers</code>	Echo the current version and copyright information to <code>stderr</code> . This is the same header that appears at the top of created makefiles. If you use this option with no other parameters, the information echoes and CreateMake exits. Otherwise, the information echoes and processing continues.

Usage You do not usually call CreateMake directly; instead, you should use `Makeit` to automatically invoke it (see “Makeit” on page 28). `Makeit` executes CreateMake if the makefile is out-of-date or missing.

Makefile format CreateMake generates a standard AIX makefile whose content depends on the *targets* in *sourcefile*. Each makefile supports the standard Taligent build steps (Includes, Objects, Exports, and Binaries).

Examples Simple projects require simple `make` commands. For example, to create a makefile named `Sample.Make` which builds a *target* from the C source files in the working directory:

```
CreateMake Sample.PinkMake > Sample.Make
```

FINDSYMBOLS


FindSymbols reports the shared libraries that contain the specified symbols.

Syntax FindSymbols ['*symbol*' ...]

Arguments '*symbol* The mangled, demangled, or mixed-form symbol to locate. The argument must be enclosed in single quotes (').

Usage Use FindSymbols when MakeSharedLib or MakeSharedApp report unresolved symbols, and you want to know which libraries you should add to the link list in your *.PlinkMake file.

The first time you run FindSymbols, it builds a cache file: \$TaligentExport/_AllSymbols. Subsequent runs consult that cache file. To rebuild or update the file, delete it and rerun FindSymbols. When you install a new build, InterimInstall should delete the cache.

 **NOTE** If FindSymbols can't locate a symbol that you are certain exists, the symbol is probably an inline. There is no way to find inlines, because they are compiled into client code, as opposed to being compiled into and exported from a library for use by clients.

Because the implementation of an inline must be compiled with the header, you should be able to find the inline declaration if you do enough searching: it will either be hidden down near the bottom of the header, or in another file that is an #include in the header (typically similar to "XXXXImplementation.[ih]").

A compiler is free to not inline an inline if doing so would generate worse code. This means that some symbols declared inline might not actually be inlined, and so can wind up compiled into and exported from a library which—if not in the *.PlinkMake's link list—would lead to an unresolved symbol error.

Example You will typically use FindSymbols to locate the library that caused an "Undefined symbol" error when your build fails. For example, Makeit might list errors for undefined symbols encountered when MakeSharedLib executes. In the messages, look for errors like these below the MakeSharedLib command line:

```
... MakeSharedLib -o JustAViewLib ...  
ld: 0711-317 ERROR: Undefined symbol: .TGArea::~TGArea()  
ld: 0711-317 ERROR: Undefined symbol: .TRGBColor::~TRGBColor()  
ld: 0711-317 ERROR: Undefined symbol: .TGRect::~TGRect()  
ld: 0711-317 ERROR: Undefined symbol: __vtt12TContentView
```

To find the library files in which these symbols are defined, run `FindSymbols` and specify the symbol exactly as it appears in the error listing. The symbol name must be enclosed within apostrophes (single quotes).

```
FindSymbols '.TGArea::~~TGArea()'
```

Which produces a listing like this:

```
TGArea::~~TGArea():
    HighLevelAlbert
```

This is the unique set of libraries identified:

```
HighLevelAlbert
```

*Link tag to add to
your *.PinkMake*

This listing indicates that the symbol is in `HighLevelAlbert`.

To look for multiple symbols at once, include each as a separate argument on the `FindSymbols` command line:

```
FindSymbols '.TRGBColor::~~TRGBColor()' '.TGRect::~~TGRect()' '__vtt12TContentView'
```

Which produces this listing:

```
TRGBColor::~~TRGBColor():
    LowLevelAlbert
TGRect::~~TGRect():
    CommonAlbert
    HighLevelAlbert
    __vtt12TContentView:
    NewGraphicApplicationLib
```

This is the unique set of libraries identified:

```
CommonAlbert
HighLevelAlbert
LowLevelAlbert
NewGraphicApplicationLib
```

Notice that `TGRect::~~TGRect()`: appears in `CommonAlbert` and `HighLevelAlbert`. When you get multiple libraries, you probably need to include only one. Try one and if you still get errors for the symbol, try the other. In a worst case, include both. This example only needed `HighLevelAlbert`.

It's also possible to find symbols before using `Makeit`. To do so, you must take a symbol from C++ code and put it into the canonical form used by the linker. This isn't easy. Here are some rules for functions that work 80-90% of the time:

- 1** Remove the return value.
- 2** Preface the function with the *ClassName* followed by "::".
- 3** Remove all argument names.
- 4** Remove all whitespace, except:
 - A** There should be exactly one blank after all `const` keywords inside a function's argument-parenthesis.

- ⑨ There should be exactly one blank after a function's closing ')' and before a const keyword.

For example:

```
class TSomeClass {  
    int SomeFunc( const TSomeType* someArg,  
                 TOtherType& otherArg ) const;  
}
```

becomes:

```
TSomeClass::SomeFunc(const TSomeType*,TOtherType&) const
```

Complications creep in when one or more of the types involved are #define's or typedef's. In such cases, it's better to choose a different function.

With practice, you can get good at this technique, and can even find other kinds of symbols (enum's, for example). This may seem like a lot of work, but at least you don't have to keep running the linker.

This technique is best when you have a program that is already compiled and working, and you add some new functionality to it. Then you have a good idea of what new symbols you've introduced, and what symbols to search for.

IPCPURGE

IPCPurge purges global shared interprocess resources (such as global semaphores and shared segments) from memory. Usually IPCPurge is called from mop, which is called from StopPink.

Syntax IPCPurge



CAUTION IPCPurge causes running Taligent applications to end abnormally.

MAKEEXPORTLIST

MakeExportList generates an .e file from an .o file (which is a combination of one or more x1C compiled .C files). Clients of a shared library link with the .e file, which is a text list of all the symbols that the shared library provides.

Usage CreateMake executes this command for you when you are building libraries. You should not have to run it independently.

Example MakeExportList -l SharedLib MyLib.o > SharedLib.e

MAKEIT

Makeit is a wrapper (a front end) to make. Makeit simplifies the builds and provides consistency. It has the ability to traverse project hierarchies and convert makefile descriptions to real makefiles (by calling CreateMake).

Installation

MakeIt is located in `/usr/taligent/bin` and requires no installation. Make sure this directory is in your command search path if MakeIt fails to run.

Makeit has only a few options; however, it passes all other options onto make. So in effect, Makeit has the same options as make, plus its own options.

Syntax

Makeit [*options*] [*Targets*]

Makeit passes any unrecognized arguments on to make.

Arguments

<code>-c</code>	Do not build subprojects. By default, Makeit operates recursively on subprojects from the bottom up, executing targets at every project it finds in a <i>subproject</i> block.
<code>-D</code>	Do not rebuild a make file, even if it is out of date.
<code>-i</code>	Do not stop when errors are encountered. This is passed on to make as <code>-i</code> .
<code>-fast</code>	CreateMake option; Makeit passes this option to CreateMake.
<code>-M</code>	Force all makefiles to be rebuilt on the fly by calling CreateMake even if files are up-to-date.
<code>-T</code>	Traverse the project tree, but do not build anything.
<code>VAR=value</code>	Assign <i>value</i> to the variable named <i>VAR</i> . Makeit passes this expression to make to alter makefile variable usage.
<code>-vers</code>	Echo the current version and copyright information to <code>stderr</code> .
<i>Targets</i>	The targets to build. If you omit this option, Makeit builds each target in the current project (Includes, Objects, Exports, and Binaries) and the necessary dependencies. You can also specify <code>complete</code> to build the four targets. Makefiles is a special <i>target</i> that generates a new makefile, but does not build anything. Use this for debugging. Makeit Makefiles

Usage	<p>Go through each build process target (Includes, Objects, Exports, and Binaries) and build the necessary dependencies.</p> <pre>Makeit</pre> <p>To build DemoApp, and its subprojects:</p> <pre>Makeit DemoApp</pre> <p>A common mistake is to tell Makeit to build one target (like the previous example), and not realize that it will execute <code>make DemoApp</code> on all subprojects—many of which do not have a target DemoApp. To prevent Makeit from building subprojects:</p> <pre>Makeit -c DemoApp</pre> <p>To require Makeit to execute only the Includes and Exports targets in each directory.</p> <pre>Makeit Includes Exports</pre>
Passing options to make	<p>Makeit accepts (and passes) all options to make. You can use this feature to pass options to make. For example if you want make to continue building even if an error occurs (<code>-i</code> option for make):</p> <pre>Makeit -i Objects</pre> <p>This works similarly for any make option. Makeit is smart enough to pass on any arguments for options too. For example, you can override variables in makefiles as you can with make. To override the COPTS (compiler options) variable in the makefile:</p> <pre>Makeit COPTS=-g Binaries</pre>
Creating makefiles	<p>Makeit can build makefiles on the fly. Makeit rebuilds a makefile if:</p> <ul style="list-style-type: none">⌘ the *.Make file does not exist⌘ the *.PinkMake file is newer than the *.Make file⌘ you specify <code>-M</code> to override the automatic makefile generation <p>Makeit uses <code>CreateMake</code> to translate the makefile descriptions (*.PinkMake) to makefiles (*.Make).</p>
Universal.Make	<p>To prevent duplication in each makefile, and to allow for more flexibility, Makeit includes <code>Universal.Make</code> in every makefile (*.Make). <code>Universal.Make</code> contains global targets and rules, such as Includes, Objects, Exports, and Binaries.</p>

Other global targets

In addition to the global targets previously mentioned, other global targets are also useful because they are applied only to the projects in the build and not to every directory in the hierarchy. For example you might have an entire subsystem, that exists, has been checked into SCM, but is not part of the build. These targets will not be applied to those projects:

Capitalization
is optional

Global Target	Task
Clean	Removes all .o and .e files, and libraries that were built.
Complete	Expands into the four standard targets: Includes, Objects, Exports, and Binaries.
Makefiles	Allows you to traverse the directory and rebuild makefiles as needed.

MAKESHAREDAPP

MakeSharedApp builds executable applications or programs (it is a wrapper for an x1C command with special options).

Usage

CreateMake generates this command for you when you build binaries or programs (applications). You should not need to run it independently.

Example

The following example builds the MyApp executable, and specifies two search paths -L. (current directory) -L/usr/lib/dce which will be searched in the order specified to load shared libraries SharedLib1 and SharedLib2. If SharedLib1 and SharedLib2 are not in these directories, the AIX runtime searches in the path specified by LIBPATH.

```
MakeSharedApp -o MyApp AppMain.o SharedLib1.e SharedLib2.e -L. -L/usr/lib/dce
```

MAKESHAREDLIB

MakeSharedLib is a wrapper to the AIX makeC++SharedLib script, which combines .o and .a files into a single shared library, and uses .e files to resolve external symbols located in other shared libraries.

Usage

CreateMake generates this command for you when you are building libraries. You should not have to run it independently.

Example

To create a shared library named *SharedLib1* that uses the code in *MyLib.o*, and resolves external symbols by looking in *SharedLib2.e*:

```
MakeSharedLib -p 6000 -o SharedLib1 MyLib.o SharedLib2.e
```

MAKESOL

MakeSOL registers export-file libraries for Taligent Application Environment.

Syntax MakeSOL [-c | -t | -e *pattern* | -i *pattern* | -I *files* | -E *files*] [-a *file*] [-v]

Arguments	-a <i>file</i>	An additional file to register.
	-c	Detects linking against .e files that don't have corresponding library files.
	-e <i>pattern</i>	Excludes files matching the pattern.
	-E <i>file</i>	Excludes the files listed.
	-i <i>pattern</i>	Includes files matching the pattern.
	-I <i>file</i>	Includes the files listed.
	-t	Includes the test libraries. By default, they aren't included.
	-v	Lists—to <code>stdout</code> —status messages and the files registered. If you omit this option, only warning and error messages appear.

Usage Use MakeSOL to add new libraries; ones that aren't already in the build.

MOP

`mop` is a wrapper for `IPCpurge`. In addition to calling `IPCpurge`, it removes temporary files created by the AIX implementation of `ScreamPlus`. You can run `Mop` independently, but it is best to let `StartPink` or `StopPink` call it.

Syntax `mop`

RUNDOCUMENT


RunDocument creates, opens, or deletes a document that accesses a shared library already running in the Taligent Application Environment workspace.

Syntax RunDocument [-c *Class SharedLib* | -o [-s *Mode*] [-p *Way*] | -d] [*DocumentName*]

Arguments	<p>-c <i>Class SharedLib</i> Creates a new document from the TAbstractDocumentStationery subclass <i>Class</i>, which is defined in the shared library <i>SharedLib</i>. Can be combined with -o to open and create at the same time.</p> <p> If <i>DocumentName</i> already exists, RunDocument appends an integer <i><n></i> to the name, where <i><n></i> is 2 or greater such that the name is unique.</p> <p>-d Deletes <i>DocumentName</i>.</p> <p>-o Opens <i>DocumentName</i>. Can be combined with -c to open and create at the same time.</p> <p>-p <i>Way</i> Specifies the task in which to open the document. <i>Way</i> can be:</p> <p> 0 = open in same task (default.).</p> <p> 1 = open in a new task.</p> <p>-s <i>Mode</i> Specifies the mode in which to open the document. <i>Mode</i> can be:</p> <p> 0 = examine store (default.).</p> <p> 1 = assume this is a basic document.</p> <p> 2 = assume this is a compound document.</p> <p><i>DocumentName</i> The document created, opened, or deleted. If you omit <i>DocumentName</i>, use "Untitled" as the default.</p>
-----------	---

Usage RunDocument prints, to stdout, one of these status codes:

0	No error.
1	Syntax error in arguments.
2	Stationery class not found.
3	Document not found.
4	Could not delete document.
5	Could not open document.
6	Could not determine document store type.

 **NOTE** In SDK1, if you are running multiple instances of RunDocument, two of them can pick up the same document name. One will successfully create that document, but the other will get an exception that causes a SIGIOT. Be sure to use a unique name for each instance.

SHAREDLIBCACHE


SharedLibCache builds a cache of symbol addresses at the end of shared libraries for fast subroutine lookup during TStream::Flatten and TStream::Resurrect. MakeSharedLib uses SharedLibCache to cache the default constructors of MCollectibles for resurrection.

Syntax SharedLibCache [-d *sharedLib*] [-da *sharedLib*] [-r *sharedLib*]

Arguments

-d <i>sharedLib</i>	Create cache of symbols required for flatten/resurrect.
-da <i>sharedLib</i>	Create cache of all formal symbols (rarely used).
-r <i>sharedLib</i>	Display the contents of an existing cache.

Usage Running strip on a shared library destroys its cache; rerun SharedLibCache to rebuild the cache.

 NOTE SharedLibCache is also called slcache.

SLIBCLEAN

slibclean cleans up global semaphores and global variable space. (Run by StopPink.)

Syntax slibclean

Usage Run slibclean between running different versions of Taligent Application Environment. The file /etc/slibclean should be owned by root and swid.

SMARTCOPY

SmartCopy is a cp imitator that solves one specific problem: during the Includes phase of the build, when header files are copied to \$TaligentIncludes, if a file exists in \$TaligentIncludes, and it is write protected, cp fails but SmartCopy does not. SmartCopy performs one other important task: it preserves the modification date to prevent unnecessary rebuilds. SmartCopy copies a file unless the target file has exactly the same date and time, and the same size as the source file. This should save you the time of copying the same file over itself, and is more certain to copy a file that is truly different.

Syntax SmartCopy sourceFile... destFile

Arguments	<i>destFile</i> The destination of the file being copied.
	<i>sourceFile</i> The file(s) to copy.

STARTPINK

StartPink starts the Taligent AIX reference layer and several servers. The remaining servers are started when they are needed (when you launch a Taligent Application).

Syntax StartPink [-a *applicationName*] [-q] [-n [-s]]

Arguments	-a <i>applicationName</i> Load and run the named application.
	-n Use merged servers. If you omit this option, StartPink uses non-merged servers. Merged servers give you a smaller memory footprint, faster start-up, and better interactive performance, but less stability.
	-q Do not load shared libraries.
	-s Start merged servers as a one. If you omit this option, the merged servers start in three groups. -s has no effect if you omit -n.

Usage When the StartPink script finishes, it displays a message, similar to this:

```
Welcome to the Taligent AIX Layer
Based from v1.0d29
```

```
Copyright (C) 1993, 1994 Taligent, Inc.
All rights reserved.
```

STOPPINK

StopPink safely takes down the Taligent AIX layer. StopPink seeks out and kills the servers that StartPink started. It also runs mop and slibclean, see “mop” on page 31.

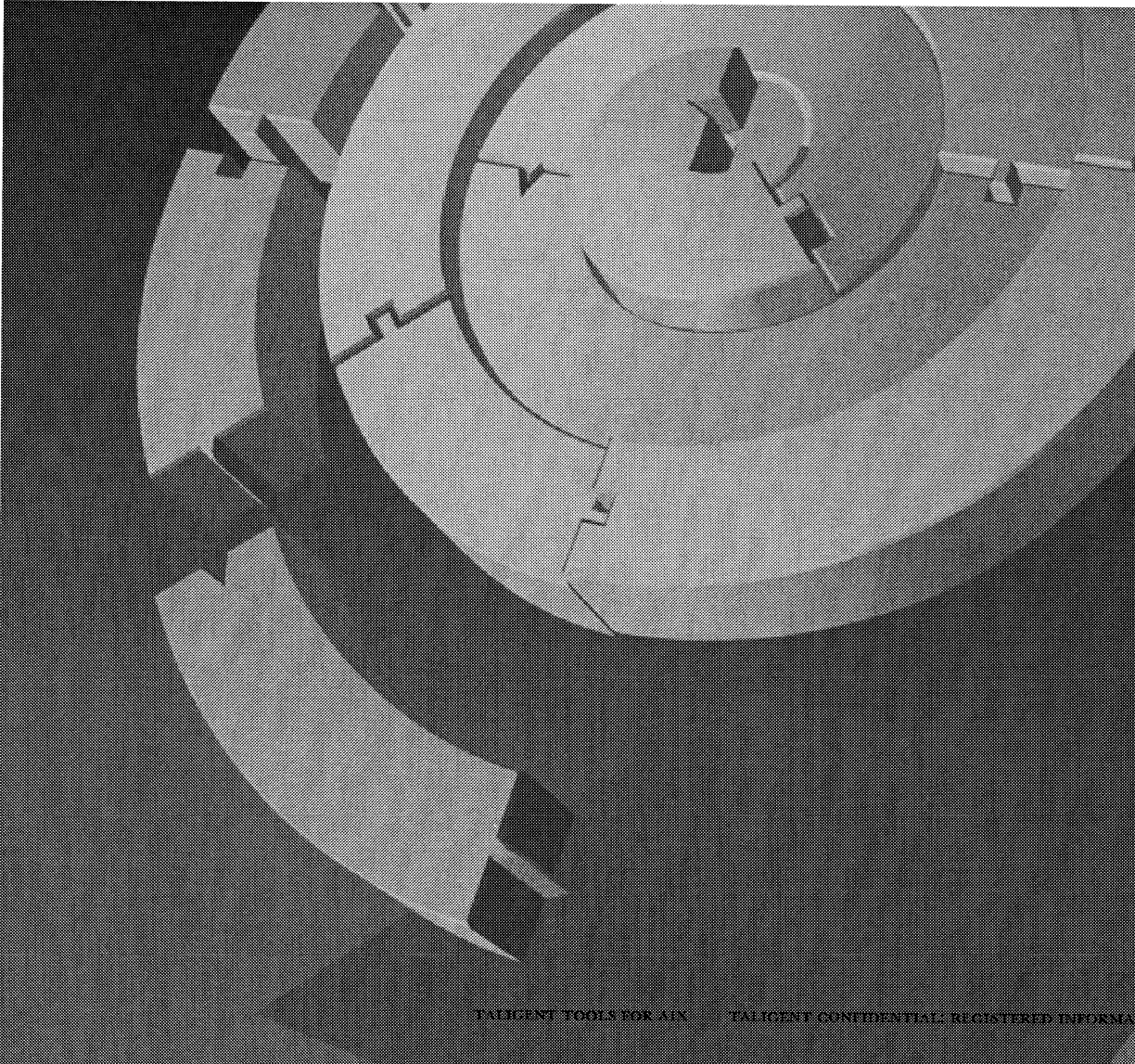
Syntax

StopPink

Usage

StopPink only kills system servers and applications, *not* applications that are running. Always quit your applications before running StopPink.


StopPink



CHAPTER 4

CREATEMAKE

CreateMake generates *.Make files for use with the Taligent build system. This chapter describes each of the targets, tags, and options that are available for input into CreateMake. For information about using CreateMake, see “Makefiles” on page 9.

 **NOTE** When building for Taligent Application Environment, references to compile and link methods are referring to the IBM x1C compiler and linker.

CreateMake is a Taligent AIX tool that evolved from a similar Macintosh tool called CreatePinkMakeFile. CreateMake is faster and can perform more operations than its predecessor. Also, CreateMake does not require external tools, such as the old MakeMake. CreateMake accepts most of its predecessor's keywords; however, these keywords are not implemented:

asmoption, dependson, exportclient, exportsample, ISR, makemakeoption, opusbugtemplate, otherheaderdir, othersourcedir, plinkclientoption, plinklibraryoption, plinkoption, prelude, programdata, and resident.

Keyword categories

There are four categories of CreateMake keywords:

Targets generate dependencies for a specific output target. All targets contain at least one source file declaration with which to build the target. *Targets can contain various tags, but never other targets.*

Tags are target specific identifiers for components within that target. Use tags within targets to specify, for example, source and header files.

Variables are keywords used within the generated makefile to control various options.

Customs are keywords that allow custom control over the generated makefile. `start` and `end` are examples of custom keywords.

Path names

If a file name contains a slash or starts with a variable, such as `$(...)`, CreateMake assumes that you have specified a complete file name. To interpret the name literally, enclose the name in single quotes; that is, CreateMake will not prepend a directory or append a suffix.

APPLICATION

This is an obsolete target; use `binary` instead.

BINARIESSUBFOLDERDIR

This variable overrides the default destination for binaries built by the makefile that CreateMake generates. The default directory is `$(TaligentBinaries)`.

Syntax

`binariessubfolderdir: directoryPath`


Argument

directoryPath

The path location to copy the built binaries to. This can be an explicit path or a shell variable.

Example

```
binariessubfolderdir: $(TaligentRoot)/MyBinaries:
library MyLibrary {
source:
    Library.c
}
```

 **NOTE** For Release A, this keyword is a synonym for `subfolderdir`, the directory identifier used by `export{subfolder:}`. In later releases, this variable will work as described.

BINARY

This target creates dependencies for a Taligent application, generates all make dependencies for creating a Taligent application, and builds an executable/library pair with all sources in the library.


Syntax `binary name {
}`

Argument `name` The name of the target, and the name used as a prefix for makefile variables, include lists, and dependencies.

An unsupported version of this target is available with the `ubinary` keyword. Unsupported targets are similar to supported targets, except that they are not built in the normal build process (`Makeit`) and require the desired target to be explicitly stated for the build to occur.

Example Produce a makefile for compiling the three source files, link them together with standard Taligent libraries, and create a main program binary and a shared library containing most of the code. Both of which contain the name “MyApp”:

```
binary "MyApp" {  
source:  
    main.c  
    TMyApp.c  
    TMyView.c  
}
```

 NOTE `program` is a synonym for `binary`.

BUILD

This tag is for specifying build rules that control a specific target, from within that target. The lines following `build` must have the correct indentation; they are copied directly into the generated makefile.

Syntax

```
build:
"$(ObjDir)/Sample.op" : Sample.txt
    $(BuildHelp) Sample.txt -o target
```

Example

```
libraryMySample {
source:
    SampleStartup.c
    SampleIndex.c

build:
"$(ObjDir)/Sample.op" : Sample.txt
    $(BuildHelp) Sample.txt -o target

link:
    Sample.op
}
```

COMPILEOPTION

This variable sets a local variable in the makefile that is used in any compile commands executed.

Syntax

```
compileoption: -d option
```

Argument


option Any option you want to pass on all compile command-lines generated.

Examples

Create a parent object that requires one source file. Pass `_WHATEVER_` to the compiler when that source file is compiled:

```
compileoption: -d _WHATEVER_

parentobjects MyObject{
source:
    HandleObject.c
}
```

 **NOTE** `cpluseption` is a synonym for `compileoption` that will soon be eliminated. Change all occurrences of `cpluseption` to `compileoption`.

DEVELOPMENTOBJECT

This target combines the specified source files into a library object, and copies the result object file to `$TaligentDevelopment`.

Syntax


```
developmentobject name {  
}
```

Argument

name The name of the target.

Examples

```
developmentobject "SampleObject" {  
source:  
  SampleInput.c  
  SampleOutput.c  
}
```

 NOTE `developmentobject` is currently treated the same as `object`.

END

This custom target allows you to supply a block of make commands to copy into the end of the generated makefile.

Syntax

```
end {  
  makeCommands  
}
```

Argument

makeCommands Any valid makefile syntax. CreateMake places this block directly into the generated makefile; pay careful attention to indentations and syntax.

Example

```
end {  
Foo : Bar  
  #build rules  
}
```

EXPORT

A variable that specifies that files in your project be exported to various Taligent directories.

Syntax

```
export {
    exportTags
}
```

Argument

<i>exportTags</i>	Control which files are exported. Valid tags are: binary, client, subfolder, program, data, script, server, library, testlibrary, testdata, and script.
-------------------	---

Example

The example shows the destination of each of the supported tags.

```
export {
binary:
    SampleExportBinary           // to $(TaligentBinaries)
client:
    SampleExportClient          // to $(TaligentLibraries)
subfolder:
    SampleExportSubfolder       // to $(TaligentBinaries)/subfolder
program:
    SampleExportProgram         // to $(TaligentBinaries)
data:
    SampleExportData
script:
    SampleExportScript          // to $(TaligentSamples)
server:
    SampleExportServer          // to $(OPD)/Servers:
library:
    SampleExportExportLibrary   // to $(OPD)/SharedLibraries:
testlibrary:
    SampleExportTestLibrary     // to $(OPD)/SharedLibraries:
        TestSharedLibraries:
testdata:
    SampleExportTestData        // to $(TaligentTests)TestData:
testscript:
    SampleExportTestScript      // to $(TaligentTests)TestScripts:
}
```

HEADER

Header files listed after this tag specify an explicit dependency for the target.


Syntax header:
 headerFiles

Argument *headerFiles* The header files on which the target is dependent.

Examples

```
library MyLibrary {
source:
    LibraryInit.c
    LibraryIO.c

header:
    $(CustomHeaders)Library.h
}
```

 **NOTE** In Release A, `header` acts like `publicheader` in that the specified files are exported to `$TaligentIncludes`. `header` will act as described in future releases.

HEADERDIR

This tag specifies an alternate directory in which header files are stored. By default, CreateMake generates makefiles with references to headers in the same directory as the makefile. CreateMake passes the reference to the compiler.

Syntax headerdir:

Example headerdir: ../MyHeaders:

HEAPSIZE

This target controls the allocated heap size of a built Taligent application.

Syntax

```
heapsize: heapSize
```

Argument

<i>heapSize</i>	The size, in bytes, of the heap.
-----------------	----------------------------------

Example

```
binary QECalc {
source:
    Main.c
heapsize: 1000000    // 1,000,000 bytes
}
```

LIBRARY

This target creates dependencies and makefile commands for creating an library to be used in the Taligent runtime system.

Syntax

```
library name {
}
```

Argument

<i>name</i>	The name of the target.
-------------	-------------------------

Examples

```
library "MyLibrary" {
source:
    LibraryInit.c
    LibraryIO.c
}
```

LINK

This tag specifies all files to link within a target.

Syntax

```
link:
    linkFiles
```

Argument

<i>linkFiles</i>	These files are linked with the listed source files and any other object listed in the target. If you omit this tag, nothing is explicitly linked in, and \$UniversalLinkList is used.
------------------	--

Example

This example produces a Taligent program (see “binary” on page 39) by linking with the files `MenuLib` and `WindowLib`, in that order.

```
binary MyProgram {  
source:  
    main.c  
    Test1.c  
link:  
    MenuLib  
    WindowLib  
}
```

LOADDUMP


This target creates build rules for creating a loaddump file with the specified headers. All targets built in a *.`PinkMake` file will have dependencies on the specified loaddump file.

Syntax

```
loaddump loadDumpFilePath {  
}
```

Argument

loadDumpFilePath The path of the loaddump file. If this file does not exist during the build's objects phase, the build creates this file.

 **NOTE** This syntax is not supported when building for Taligent Application Environment until the AIX development environment supports loaddump files.

Example

Create a loaddump file called `MyProject.Dump` in the directory pointed to by `$(TaligentRoot)/Dumps`: with the given header files included in it. The header files must be valid files in `$(TaligentIncludes)` or `$(TaligentPrivateIncludes)`.

```
loaddump "$(TaligentRoot)/Dumps/MyProject.Dump" {  
    Application.h  
    Test.h  
    Format.h  
    Dialogs.h  
}
```

LOCAL

See the description of “localheader.”

Syntax `local:`

LOCALHEADER

This tag specifies header files to export to the `localheaderdir` header directory.

Syntax `localheader:`
 `headerFiles`

Argument *headerFiles* The files to export to the `localheaderdir` directory.

Examples Export the file `Parents.h` into a directory called `:LocalHeaders:`. If you omit the tag `localheaderdir`, the file is copied to the current directory.

```
localheaderdir: ./LocalHeaders:
```

```
parentobject MyParentObj {
source:
    Parent1.c
    Parent3.c
    Parent5.c
localheader:
    Parents.h
}
```

LOCALHEADERDIR

This variable specifies the directory in which to export header files for the target.

Syntax `localheaderdir: localheaderPath`

Argument *localHeaderPath* The directory in which to export local headers. if you omit this variable, the headers are copied into the same directory as the source files.

Example See the example for “localheader.”

MAKE

With this target you can specify you own build rules. Unlike `start` and `end`, the `make` target can appear anywhere in the input, and you can have multiple `make` blocks in the input.

Syntax

```
make {  
    buildRules  
}
```

Argument

buildRules Your own build rules.

Examples

```
make {  
    Foo : Bar  
        # build rules  
}
```

OBJECT (TAG)

This tag specifies a target's a dependency on object files that might be built within another target or project.

Syntax

```
object:  
    objectFiles
```

Argument

objectFiles Link these object files in after any other files produced from specified source files within the target.

Example

Create a dependency for `MyLibrary` on the file `LibI0.c.o`, which is an existing object from another target in the same project or another project. The explicit path to the object file is not required.

```
library MyLibrary {  
    source:  
        Main.c  
    object:  
        ../ObjectFiles:LibI0.c.o  
}
```

OBJECT (TARGET)

This target combines files into a single library object for later use in another target or project.

Syntax `object name {`
`}`

Argument *name* The name of the target.

Example Combine three files into a single library object called `MyObject`, and copy it to `$ObjDir`, if it is not the default.

```
object MyObject {
source:
  MySample.c
  MyOtherSample.c
  MyExtraSample.c
}
```

OBJECTDIR


This variable specifies the directory for compile output and link input (object files) built within the current project.

Syntax `objectdir: path`

Argument *path* The directory for all compile output and link input. If you omit this variable, the build stores these files within the current project in the `:ObjectFiles:` directory.

Example Change the directory for built objects to `MyObjects`, one directory up in the tree.

```
objectdir: ../MyObjects:
```

 **NOTE** In Release A, `objectdir` does nothing. This will be fixed in a later release.

PARENTOBJECT


This target is similar to the object target. It combines the specified files into a single library object, then it copies the built object into `$ParentObjectDir` as specified by the `parentobjectdir` variable.

Syntax `parentobject name {`
`}`

Argument `name` The name of the target.

Examples Create `MyObject` from the compiled output of the three specified files, then copy it to the `$ParentObjectDir` directory.

```
parentobject MyObject {  
source:  
    MySource.c  
    MyMenus.c  
    MySample.c  
}
```

 **NOTE** In Release A, `parentobject` does not export the created object to the parent directory. This will be fixed in a later release.

PARENTOBJECTDIR

This variable changes the default directory in which to copy objects built from the `parentobject` target.

Syntax `parentobjectdir: path`

Argument `path` The directory for `parentobject` targets. If you omit this variable, the target copies the files to the `ObjectFiles` directory in the parent directory.
Use only paths based on the current directory or a known directory tree. Do not use a declaration scoped to a specific user volume.

Examples Change the destination of `parentobject` copies to the `ObjectFiles` directory in a project called `Sample` in the parent directory.

```
parentobjectdir: ../Samples/ObjectFiles/
```



CAUTION Do not depend on directories that can change in other projects. In example, if the `Samples *.PinkMake` file ever has a different `$ObjDir` (set with `objectdir`), this declaration might copy the built object to the wrong place.

PRIVATE

Use this tag within a target to specify a dependency on header files located locally to the project.

Syntax `private:`
`headerFiles`

Argument `headerFiles` The local header files for the project. If you omit a header file, the build searches for the file in the local directory, then in `$TaligentIncludes`, followed by `$TaligentPrivateIncludes`. When you include a header file, the build searches in the local directory only.

Examples

```
parentobject MyObject {
source:
    MySource.c
    MyMenus.c      // Look for MyMenus.h locally, then in the other directories
    MySample.c
private:
    MySource.h     // In local directory only
    MySample.h     // In local directory only
}
```

PRIVATEHEADERDIR

This variable points to a directory to search for header files not in the source directory.

Syntax `privateheaderdir: path`

Argument `path` An optional directory for the compiler to search for header files not in the source directory.

Example `PrivateHeader.h` is not in the current directory. Without the reference to its location, compilers cannot locate it if `main.c` tries to include it.

```
privateheaderdir: ../PrivateHeaders:

library MyLibrary {
source:
    main.c
header:
    PrivateHeader.h  // not in source directory
}
```

PROGRAM

This is an obsolete target; use `binary` instead.

PUBLIC

This tag specifies which target headers the `public` tag can export to `$(TALIGENT_INCLUDES)`.

Syntax

```
public:  
    headerFiles
```

Argument

headerFiles The header files that can be exported.

Examples

Create a dependency for `MyLibrary` on the file `LibIO`. as usual. During the build's Includes phase, export this file to `$(TALIGENT_INCLUDES)`.

```
library MyLibrary {  
    source:  
        main.c  
        LibIO.c  
    public:  
        LibIO.h  
}
```

SERVER

This target creates dependencies for a Taligent application. All make dependencies for creating a Taligent application will be generated for you. This target builds a single executable with all sources linked in

Syntax `server name {`
`}`

Argument *name* The name of the target, and the name used as a prefix for makefile variables, include lists, and dependencies.

An unsupported version of this target is `userver`.

Examples Produce a makefile for compiling the three source files, link them together with standard Taligent libraries, and create a main program binary and a shared library containing most of the code. Both of which contain the name "MyServer".

```
server "MyServer" {
source:
    main.c
    Server.c
    ServerView.c
}
```

SOURCE

This tag specifies source files within targets. The order of the files in the list is the order used to compile, link, and export.

Syntax `source:`
`targets`

Argument *targets* The target files.

Examples `binary "MyApp" {`
`source:`
`main.c`
`TMyApp.c`
`TMyView.c`
`}`

SOURCEDIR

This variable specifies the directory to search for source files.

Syntax

```
sourcedir: path
```

Argument

path

The directory for source files. If you omit this variable, the build searches in the same directory as the *.Make file.

Base this path name on the current directory; do not rely on specific volume names or base directory paths—they can change from user to user.

Examples

Change the default location of source files to a directory called Source within the current project.

```
sourcedir: /Source
```

START

This custom target allows you to supply a block of make commands to copy into the beginning of the generated makefile.

Syntax

```
start {  
    makeCommands  
}
```

Argument

makeCommnds

Any valid makefile syntax. CreateMake places this block directly in the generated makefile; pay careful attention to indentations and syntax.

Examples

```
start {  
  Foo : Bar  
    #build rules  
}
```

SUBFOLDER

This tag identifies files within the export target to export to the `$SubfolderDir` within `$TaligentBinaries`.

Syntax `subfolder:`
 `exportFiles`

Argument `exportFiles` The files to export.

Examples Export to the specified files to `/MySamples/` within the `$TaligentBinaries` path.

```
subfolderdir: /MySamples

export {
subfolder:
    MySampleStuff
    MyOtherSampleStuff
}
```


SUBFOLDERDIR

This variable specifies the subfolder that is copied to from within an export block.

Syntax `subfolderdir: directory`

Argument `directory` The directory to receive export files.

Examples See example for “subfolder.”

 **NOTE** In Release A, `binariessubfolder` is a synonym that acts the same as `subfolderdir`. In later releases, `binariessubfolder` will not be a synonym. See the “`binariessubfolderdir`” on page 38 for more information.

SUBPROJECT

This target specifies subprojects to be included when the build system recursively builds directories. CreateMake places these subproject names in the `$SubProjectList` variable in `*.Make` files.

Syntax

```
subproject {  
    subProjects  
}
```

Argument

`subProjects` The sub projects to build.

Examples

Generate the `*.Make` file with the three specified subproject/directory names in the `$SubProjectList`, and allow the build system to recursively execute the `*.Make` files in each of these subprojects whenever a make is done on is project.

```
subproject {  
    FancyText  
    FancyDraw  
    FancyPrint  
}
```

TESTAPPLICATION

This target is similar to the `binary` target, but only gets built if “Makeit testing complete” is used. See “binary” on page 39 for more information.

Syntax

```
testapplication name {  
}
```

TESTLIBRARY

This target is similar to the `library` target, but only gets built if “Makeit testing complete” is used. See “library” on page 44 for more information.

Syntax

```
testlibrary name {  
}
```

TESTPARENTOBJECT

This target is similar to the `parentobject` target, but only gets built if “Makeit testing complete” is used. See “`parentobject`” on page 49 for more information.

Syntax

```
testparentobject name {  
}
```

TESTSERVER

This target is similar to the `testserver` target, but only gets built if “Makeit testing complete” is used. See “`testserver`” on page 56 for more information.

Syntax

```
testserver name {  
}
```

TOOL

This target is similar to the `binary` target. See “`binary`” on page 39 for more information.

Syntax

```
tool name {  
}
```

TRIMDEPENDENCIES

This target specifies header file paths to remove from the generated makefile.

Syntax

```
trimdependencies {  
    headerPaths  
}
```

Argument

<i>headerPaths</i>	The list of header file paths to remove from the generated makefile. If you omit these paths, CreateMake includes the list of dependencies found in <code>\$(TaligentIncludes)</code> and <code>\$(TaligentPrivateIncludes)</code>
--------------------	--

By default, CreateMake includes the list of dependent header files found in `$(TaligentIncludes)` and `$(TaligentPrivateIncludes)`. In most cases, these headers do not change and the extra dependencies result in larger make files that take longer to process. With `trimdependencies`, CreateMake removes any dependencies found in the list of header files from the generated makefile.

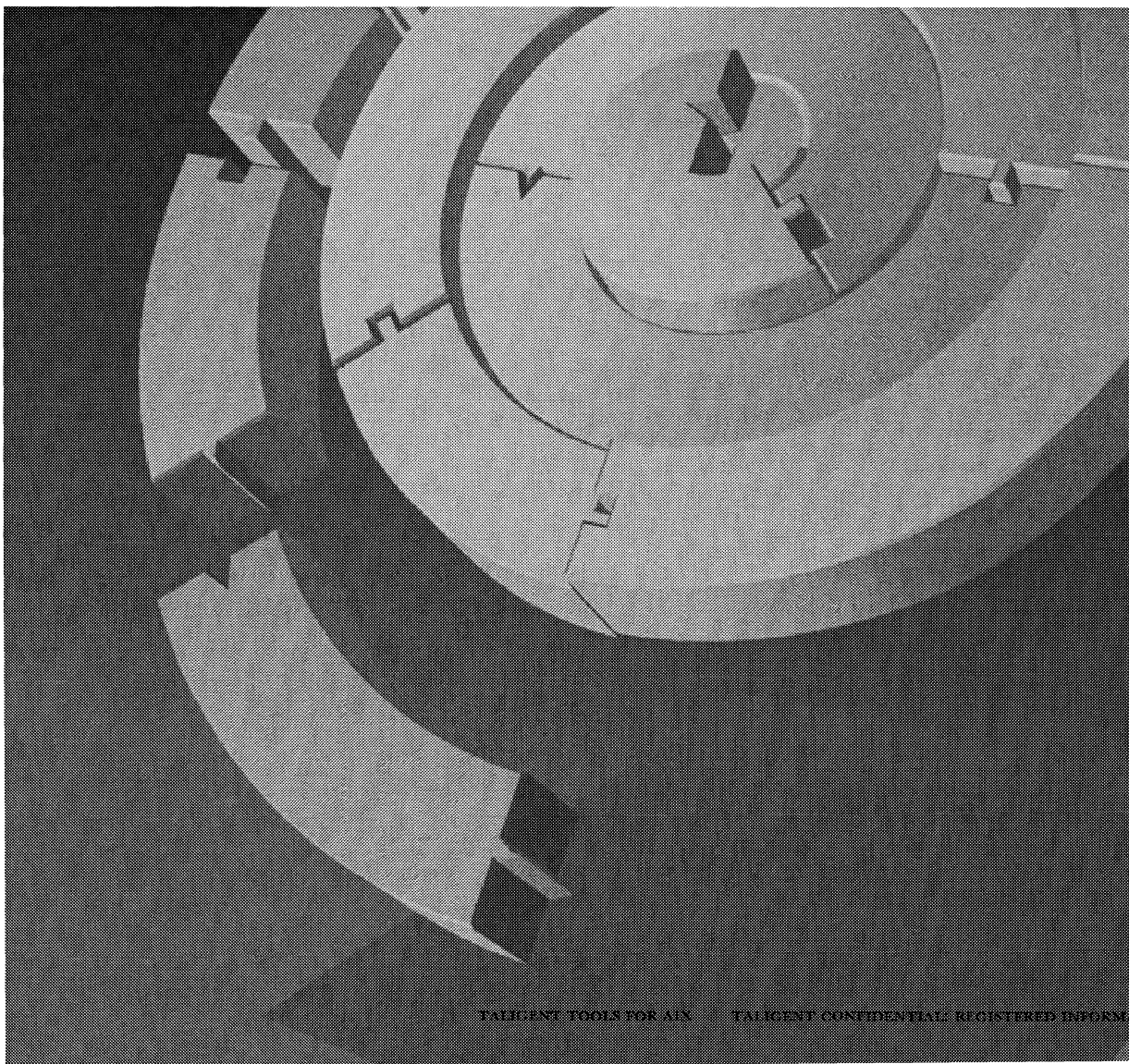
Examples

Strip out any dependencies that begin with `$(TaligentIncludes)` or `$(TaligentPrivateIncludes)`. You can do the same thing with any pathname, although generally, you only need to do this with the Taligent public and private includes.

```
trimdependencies{  
    $(TaligentIncludes)  
    $(TaligentPrivateIncludes)  
}
```



CAUTION Be careful when using this feature. If a Taligent header used by one of your source files changes, that file will not be recompiled. You must manually force the file to be recompiled.



CHAPTER 5

ANALYSIS TOOLS

The heap analysis tools are a set of applications and classes that allow you to perform heap-related debugging and dynamic analysis operations. These tools are classes that you instantiate and control dynamically, and that use TMemoryHook to receive notification of allocations and deletions in a memory heap.

The heap tools let you:

- ※ **Track block allocation** to see who allocated each block (when it is possible to follow call chains) through several levels of indirection.
- ※ **Categorize all heap blocks** to determine the type of blocks in the heap (for example, this block is a TStandardText).
- ※ **Browse heaps** to see all the blocks in the heap, with their size, type, who allocated them, who deleted them, and so on.
- ※ **Record memory usage** over time by recording the relative time of each allocation and deletion for later analysis.
- ※ **Zap memory** by filling uninitialized and deleted blocks with odd byte patterns to catch bad pointer usage errors.
- ※ **Detect heap corruption** by automatically checking the heap for corruption at each allocation and deletion.

OVERVIEW

There are two basic modes of operation:

- ⌘ **Heap monitoring** (the simplest operation) watches the heap at the event level and records allocation and deletion events. This produces an ASCII text file where each line in the file describes an *event*.
- ⌘ **Heap analysis** gathers the same data as heap monitoring, but processes the events further to produce annotated blocks within a model of the heap. It also detects anomalies in heap usage. When it stops watching, the analyzer writes a block-by-block description of the heap to an ASCII text file, where each line in the file describes a block in the heap.

Tradeoffs

Heap Monitoring	Heap Analysis
Reports each event in the heap.	Keeps and reports data for blocks currently in the heap or that were most recently deleted.
Reports more data, generates a larger data file.	Reports less data, generates a smaller data file.
Runs slower.	Runs faster.
Does not detect problems.	Detects problems, such as double deletion.

To use the local heap tools, modify your code to instantiate a heap tool object—either `TLocalHeapMonitor` or `TLocalHeapAnalyzer`. Once the object is instantiated, monitoring or analysis starts. When you destroy the object (such as if it goes out of scope) monitoring or analysis stops.

Consider a class called `TLeaksLikeASieve`, which leaks storage when its `Leak()` method is called. The following code starts monitoring, calls the suspect method, then automatically stops monitoring when the monitor object goes out of scope:

```
#include <LocalHeapMonitor.h>           // for TLocalHeapMonitor
void main()
{
    // Start monitoring; continue until object 'monitor' is destroyed.
    TLocalHeapMonitor monitor;
    TLeaksLikeASieve leaker;
    leaker.Leak();
}
```

Tools

Both the heap monitoring tools and the heap analysis tools are available as remote (monitor a different team) or local (monitor the same team). There is no separate garbage finding tool. Garbage finding is available as a function of the other tools.

TLocalHeapMonitor	heap monitoring	local team
TLocalHeapAnalyzer	heap analysis	local team

TLocalHeapMonitor and TLocalHeapAnalyzer have minimal dependencies.

Limitations

The heap tools contain these limitations:

- ※ The heap analyzer currently keeps data for only the most recently deleted heap block. As new blocks come in, old deleted block data is lost.
- ※ The heap tools consider the heap to be one logical object. In reality, the heap consists of two subheaps, the chunky and tree heaps.

TLocalHeapMonitor

The TLocalHeapMonitor constructor has several options:

```
enum EIgnoreOld { kReportOld = 0, kIgnoreOld = 1 };  
enum EZapMemory { kDontZapMemory = 0, kZapMemory = 1 };
```

```
TLocalHeapMonitor(const char* outputFileName=0,  
                  EIgnoreOld ignoreOld=kReportOld,  
                  EZapMemory zapMemory=kDontZapMemory,  
                  FrameCount depth=8,  
                  TStandardMemoryHeap* whichHeap=0);
```

- ※ **OutputFileName** specifies the name of the output file; the default is "heap_trace".
- ※ **IgnoreOld**, if set to kIgnoreOld, causes all blocks on the heap when monitoring was started to be ignored. The default shows all such blocks.
- ※ **ZapMemory**, if set to kZapMemory, causes the memory hook to fill blocks with recognizable patterns for the purpose of debugging reference-before-initialization and reference-after-deletion errors.

Uninitialized memory gets filled with the pattern 0xDEAFBEEB.

Deleted memory gets filled with the pattern 0xFEEDEAD.

- ※ **Depth** is the maximum count of functions which the stack crawls will contain. Increasing this option provides more data in some cases, but takes up more memory and slows down the tool.
- ※ **WhichHeap** specifies which heap to monitor. If unspecified, the default heap is monitored.

TLocalHeapAnalyzer

The TLocalHeapAnalyzer constructor has several options:

```
enum EIgnoreOld { kReportOld = 0, kIgnoreOld = 1 };
enum EOnlyGarbage { kAllBlocks = 0, kOnlyGarbage = 1 };
enum EZapMemory { kDontZapMemory = 0, kZapMemory = 1 };
```

```
TLocalHeapAnalyzer(const char* outputFileName=0,
                   EIgnoreOld ignoreOld = kReportOld,
                   EOnlyGarbage onlyGarbage = kAllBlocks,
                   EZapMemory zapMemory = kDontZapMemory,
                   FrameCount depth=8,
                   TStandardMemoryHeap* whichHeap=0);
```

- ❖ **OutputFileName** specifies the output file name; the default is "heap_analysis".
- ❖ **OnlyGarbage**, if set to kOnlyGarbage, causes the analyzer to list only blocks which were allocated, but not deleted. The default lists all blocks in the heap.

All other options are the same as those for TLocalHeapMonitor.

**Heap monitoring
file format**

In heap monitoring output files, each line describes an *event* that indicates that:

- ❖ A block was allocated.
- ❖ A block was deleted.
- ❖ A block was already in the heap when monitoring was begun.
- ❖ The heap was corrupted.

Here is an example of each type of event:

Thread	Time of event	Address	Size	Type	Stack crawl
2-22982	759537687555872	0xb2362718	del	TIterator	TArrayIterator...
2-22982	759537687558595	0xb2362950	12	novtbl	THybridNumber...
0-0	old	0xb24020d0	48	TLocalSemaphore	

Thread—the identifier for the thread that called new() or delete(). For old blocks, this field is 0-0.

Time of event—the time, in microseconds, of the event. Use this value to determine the order of events and to compute the time between events, such as to find the age of a block at deletion. For blocks already on the heap when monitoring starts, this field is old.

Address—the address of the first byte of the block.

Size—the size of the block in bytes. If this is a deletion event, the size field is del.

Type—the type of the block, for blocks that represent C++ objects. If the v-table pointer is NIL, this field is novtbl. If the v-table pointer is non-NIL, but it cannot be followed to a valid destructor, this field is notype. Note that only deletion events and pre-existing block events can have type information. Allocation events are always novtbl because the constructor, if any, has not been called yet.

Stack crawl—the function that called `new()` or `delete()`. For old blocks, this field is empty. The stack crawl consists of several function names, separated by '|' characters. The first function name is the innermost. It was called by the next function name, and so forth. In the example, the stack crawls have been abbreviated. A full stack crawl looks like this:

```
TArrayIterator::~~TArrayIterator()|THybridNumerals::AddFormattingPairAbsolutely(unsigned short,long)|THybridNumerals::AddFormattingPair(unsigned short,long)|THybridNumerals::CreateStandardHexNumerals()|TTieredTextBuffer::NumberFormat()  
( )|TTieredTextBuffer::operator<<(const long)|TTieredTextBuffer::operator<<(const int)|TLocalHeapMonitorTest::ShowMem(void*,long)
```

Heap analysis file format

Heap analysis output files have two sections: the *anomaly section* and the *heap dump*. In the anomaly section, any anomalies which were detected are described. See “Dynamic error detection” on page 65 for explanations of the anomalies that can be detected.

In the heap dump section, each line describes a block in the heap. By default, it displays all blocks of the heap. You can also specify to ignore old blocks, or to show only undeleted blocks. Use the latter for finding storage leaks. See “TLocalHeapAnalyzer” on page 62 for more information.

Address	Size	Type	Age	Allocation			Deletion	
				Thread	Time	Stack	Thread	Stack
0xb0c496b4	1028	TFoo	285198	2-22981	759...	TLocal...	notask	nochain

Address—the address of the first byte of the block.

Size—the size of the block.

Type—the type of the block. If the v-table pointer is NIL, this field is `novtbl`. If the v-table pointer is non-NIL, but it cannot be followed to a valid destructor, this field is `notype`. Note that only deletion events and pre-existing block events can have type information. Allocation events are always `novtbl` because the constructor, if any, has not been called yet.

Age—the block in microseconds. If the block has been deleted, this is the age of the block when it was deleted.

Allocation thread—the thread that allocated this block.

Allocation time—the time of the allocation, in microseconds. Use this to determine the order in which blocks were allocated.

Allocation stack crawl—the function that allocated this block.

Deletion thread—the thread that deleted this block, or `notask` if the block has not been deleted.

Deletion stack crawl—the function that deleted this block, or `nochain` if the block has not been deleted.

Heap corruption

Both the heap analyzer and the heap monitor detect heap corruption by calling `TMemoryHeap::Check` after each allocation event and before each deletion event. When the heap is found to be corrupt, the tool writes a message similar to the following to the output file and echoes it to the console. In heap monitor output files, an asterisk (*) precedes each subsequent line to indicate that the corrupt heap.

```
*****
***
*** Tree heap corrupt with error 5.
*** See PrivateIncludes/TreeHeapExceptions.h for enums.
***
*****
```

The message states that either the tree heap or the chunky heap is corrupt, and it specifies an error number. This number is the return value of the `Reason()` method in the `TChunkyHeapCorrupted` or `TTreeHeapCorrupted` exception object. To determine its meaning, refer to `TreeHeapExceptions.h` or `ChunkyHeapExceptions.h` in the `PrivateIncludes` directory.

Debugging heap corruption

If you have a heap corruption bug, use a heap monitor to debug it. Although the heap analyzer also notifies you of heap corruption, it does not help you pinpoint the problem. The heap monitor shows the pattern of allocations and deletions leading up to the corruption.

In order to debug the corruption, examine the event before the corruption message. If the message that the heap is corrupt occurs before any other events, you must start monitoring earlier. Starting with the code indicated by the preceding event's stack crawl, trace forward until you find the corruption. You can either read the source code or step in a debugger. The bug will usually involve violating array boundaries or misusing pointers. If you see another heap event (allocation or deletion), backup; you have gone too far.

AIX notes

On AIX, the heap tools trigger and catch segment violation signals (SIGSEGV) during the dynamic typing of blocks. Usually this will be invisible to you. However, if you run the heap tools under a debugger, it will trap the signal SIGSEGV, and you will enter the debugger that is executing the heap tool code. To avoid this, tell the debugger to ignore the signal 11, SIGSEGV. For example, in the shell, use

```
xdb -i11 Foo &
```

where `Foo` is your program's name. Within `dbx`, use:

```
ignore 11
```

DYNAMIC ANALYSIS

In processing block events, the heap tools analyze incoming data in many ways.

Dynamic typing

The heap tools attempt to determine the type of blocks in the heap (the class they instantiate). For raw block events, all allocation events have no type information because they represent unconstructed objects. Many blocks cannot be typed.

Dynamic error detection

Dynamic error detection, or *discipline*, is the programmatic detection of errors in either the heap code itself, or calls to the heap indirectly through operators `new` and `delete`.

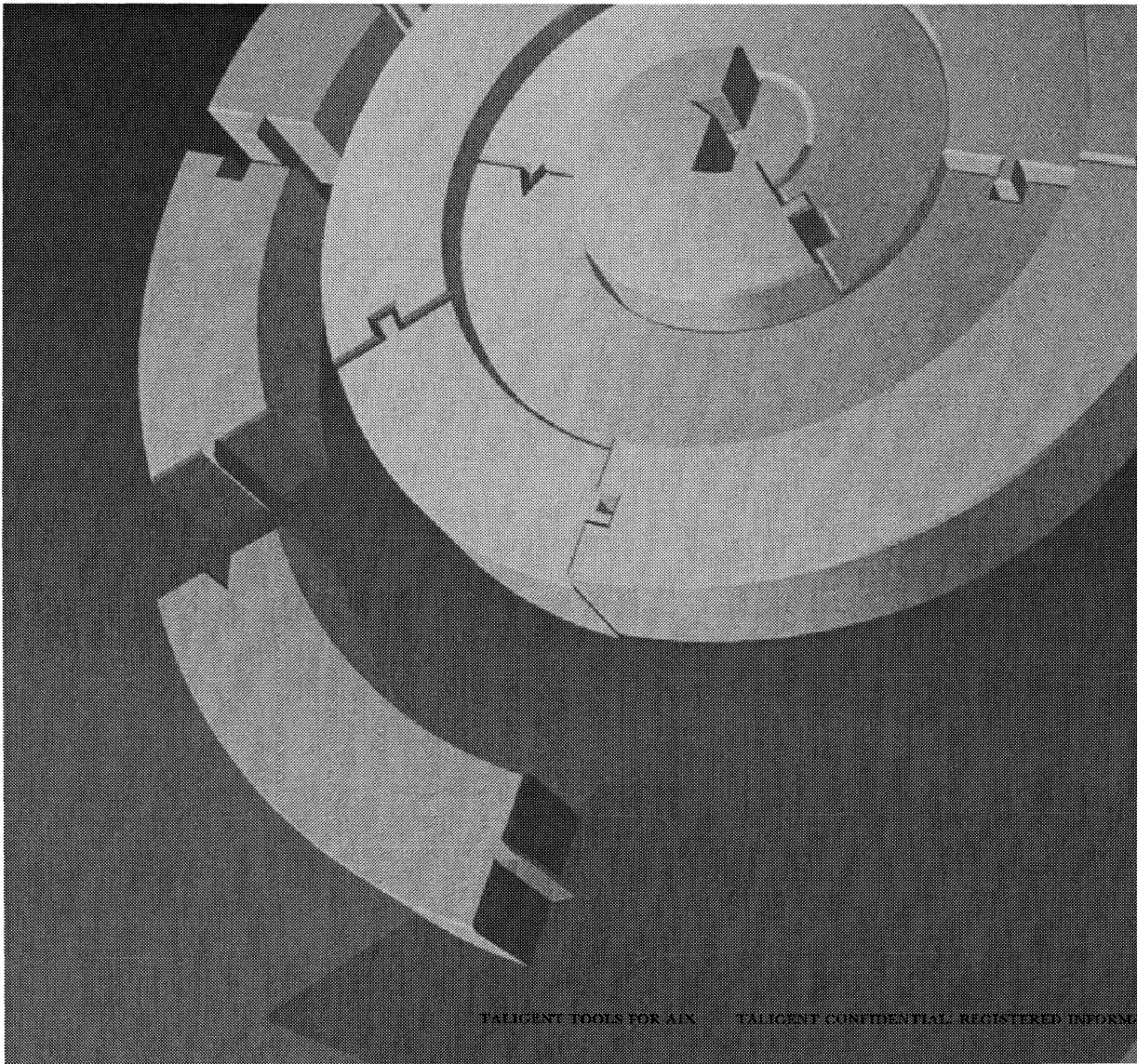
The heap model has several varieties of discipline are built into it:

Bad address deletion—the detection of addresses that do not correspond to allocated blocks in the heap. A subset of this is double deletion detection. Therefore, these two anomalies are detected by the same class in an either-or fashion.

Double deletion detection—the detection of two deletions of the same block. This is complicated by the fact that the heap allocates blocks to the same address once that address is free. The tool tracks old blocks that have been deleted. When a `delete` of the wrong type or is unmatched by a corresponding `new` occurs, it is an error.

Non-unique allocate return values—according to the *The Annotated C++ Reference Manual* (by Ellis and Stroustrup), operator `new` must return unique values (until such blocks are deleted). The tool checks this by verifying new allocations against live blocks in the existing block map.

Heap corruption—detected by calling `TMemoryHeap::Check` at each allocation and deletion.



CHAPTER 6

XcDB

Xcdb is a graphically oriented symbolic debugger for C, C++, and FORTRAN programs running under AIX Version 3, Release 2 (and later). It is a standalone program, not a windowed front-end to dbx. Xcdb has the breakpointing, stepping, and traceback capabilities common to most debuggers, but particular attention has been paid to presentation and ease of use. Xcdb understands the *name mangling* schemes used by x1C for typesafe linkage. It can display C++ class objects, display and set breakpoints in template instantiations, and display the internal contents of virtual function tables.

Xcdb runs under the X11 Release 4 (and later) windowing system and makes full use of X capabilities. Since Xcdb runs in a separate X window from the program being debugged, each has unrestricted use of the screen, mouse, and keyboard. The debugger is *mouse driven*, meaning that most interactions are performed by positioning the mouse over an appropriate screen location and clicking a key or button. Xcdb requires little or no typing.

With Xcdb, you can:

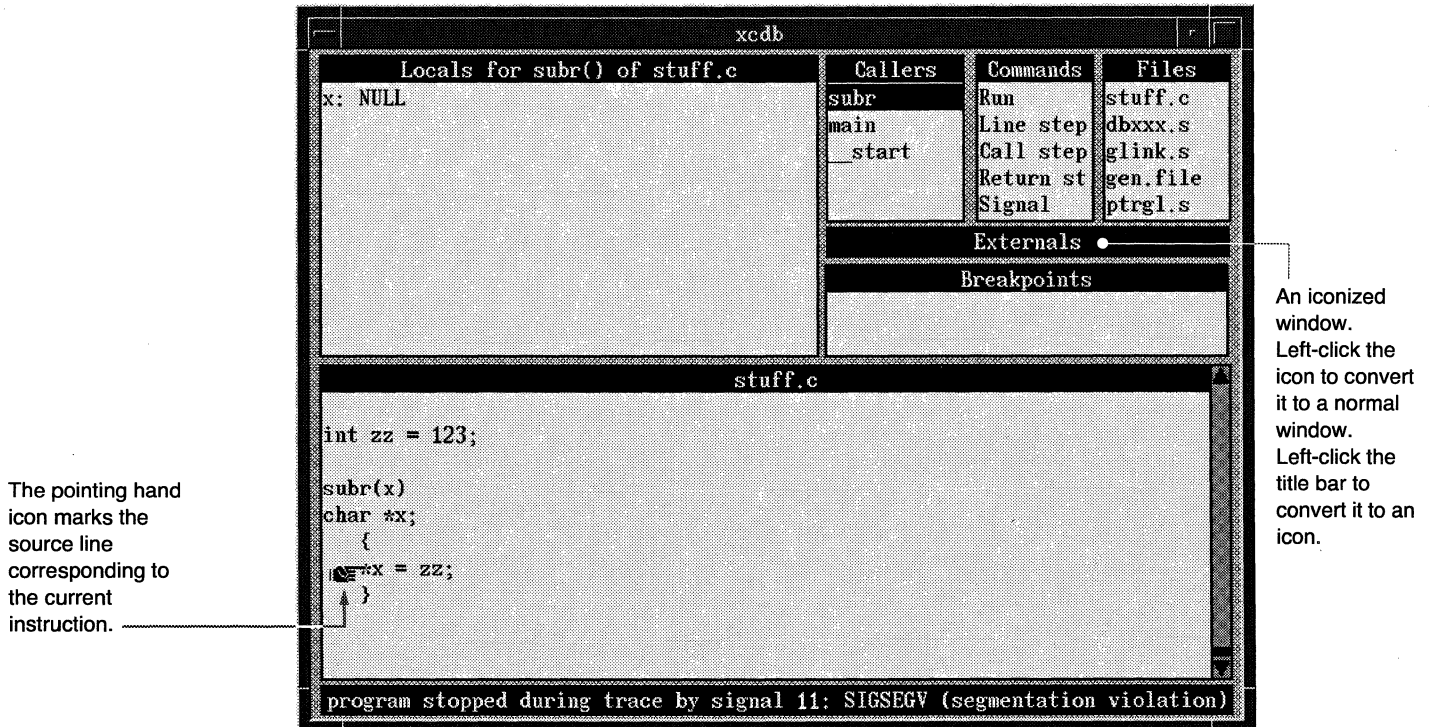
- ※ Inspect the local environment of any function in the call chain and display the format (signed, unsigned, hex, etc.) of any individual variable
- ※ Expand aggregate objects (classes, structs, unions, and arrays) to reveal arbitrary levels of detail
- ※ Tailor window layout to your preferences by making appropriate entries in your `.Xdefaults` file
- ※ Dereference pointers to reveal pointed-to objects
- ※ Obtain the type, size, and address of any object
- ※ Call upon C++ class instances to display themselves

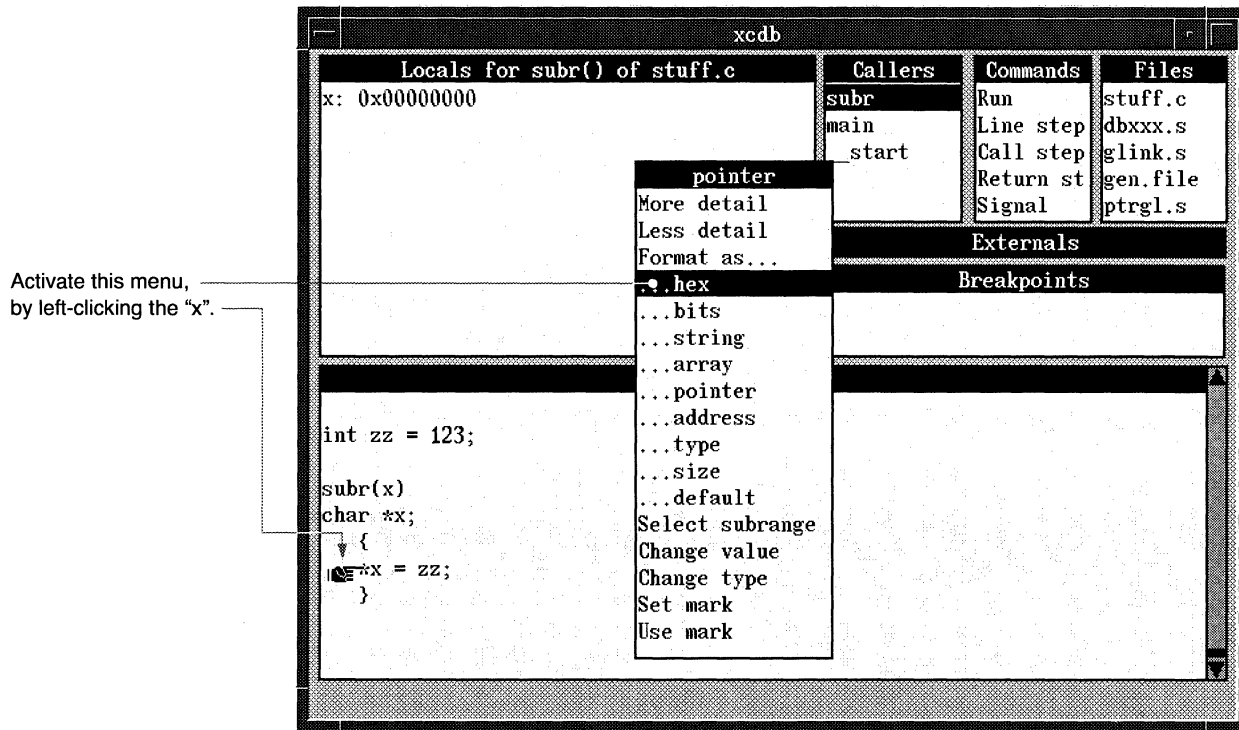
When `xcdb` traps a program interruption, either planned (by setting breakpoints) or unplanned (due to program exceptions or external signals), `xcdb` makes the program state available for inspection. The display includes window panes for:

- A traceback of uncompleted function calls
- A view of the source code for the current function, positioned at the current line
- A view of variables defined in the scope of the current function
- A view of variables defined outside the scope of any function

If the program interruption is of a type that allows execution to be continued, then you can resume program execution, perhaps after setting or clearing breakpoints. You can either ignore the signal that caused the interruption or pass it to the program.

Here is a typical display following a program exception.





SETUP

You must be running X11 Release 4 or later, with a graphics display and mouse.

Use two displays if you will be debugging programs that create virtual hft terminals (grAPHIGS programs, for example). One display should be used for X and the other for the application program.

Installation

Download `xcdb6000.tar` in as a binary file, and process it with the `tar`. For example, if you have `xcdb6000.tar` in `/tmp`, use the following commands to extract the tarfile contents into `/usr/bin`:

```

su                                # become super user
cd /usr/bin                       # go to destination directory
tar xvf /tmp/xcdb6000.tarbin     # extract contents (Xcdb)
                                  # now click Ctrl-d to become normal user again

```

`xcdb` lays out its window panes according to a predefined format. The layout is scaled to fit the window size specified by your `.xcdefaults` file, by a command line parameter, or by the window manager. “Customization” on page 86 describes how you can change the layout (and colors) to your preferences.

Signals

To be able to interrupt your program or `xcdb` asynchronously from the keyboard, define appropriate signal keys using `stty`. This document assumes that `Ctrl-c` generates an `INTR` signal and that `Ctrl-\` generates a `QUIT` signal. These are the default values on AIX systems.

Compiling

Compile *and link* the program to be debugged with the `-g` option in order to make the necessary symbolic information available. Do not use `-O` with `-g`. `xcdb` cannot reliably debug the resulting program due to code and register motions introduced by the compiler’s optimizations.

RUNNING

```
xcdb [-geometry WxH+X+Y]
      [-font fontname]
      [-title title]
      [-bw]
      [-wb]
      [-I dirname]
      [-a pid]
      [-r funcname]
      [-e numelts]
      [-c numcalls]
      [-d numdetails]
      [-b numbreaks]
      [-i signo]
      [-f fetcher]
      [-l]
      [-q]
      [-v]
      [-n]
      [-p]
      program [args...]
```

Arguments

<code>-geometry WxH+X+Y</code>	A window size and position, overriding the specification in <code>.xcdefaults</code> (if any).
<code>-font fontname</code>	The name of a font, overriding the specification in <code>.xcdefaults</code> (if any).
<code>-title title</code>	A title to place on the window border.
<code>-bw</code>	Use a black on white color scheme.
<code>-wb</code>	Use a white on black color scheme.

<i>-I dirname</i>	A directory to search for source files which cannot be found in the current directory (multiple <i>-I</i> flags are cumulative; up to 50 directories will be searched in the order listed). You can also specify the search path after Xcdb is running: see “Preferences” on page 84.
<i>-a pid</i>	The ID of an existing process to attach to, instead of starting a new process.
<i>-r funcname</i>	Specifies how far to run the program’s initialization routines. Normally the program runs to the symbol <i>main</i> , the standard starting point for C programs. To stop at some other function, specify its name. For example, to stop at the program’s first instruction, specify <i>-r \verb, __, start</i> . To stop at the function which initializes C++ static objects, specify <i>-r \verb, __, C\verb, __, runtime\verb, __, startup</i> .
<i>-e numelts</i>	The maximum count of elements to display for any array (default is 1000).
<i>-c numcalls</i>	The maximum count of functions to display in the function call traceback (default is 20).
<i>-d numdetails</i>	The count of detail levels to add (or remove) when More(or Less) detail is selected from a data object formatting menu.
<i>-b numbreaks</i>	The maximum count of breakpoints that can be set simultaneously (default is 50).
<i>-i signo</i>	The number of a signal to ignore and pass to the program (multiple <i>-i</i> flags are cumulative).
<i>-f fetcher</i>	The name of a program to call when the debugger needs to display a source file that it cannot find in the regular unix file system. The debugger invokes the program, passes it the name of the desired file as a command line argument, and display its output in the Listing window pane. Use this feature if, for example, your source files are kept in an SCCS or RCS database.
<i>-l</i>	Write window layout information to a file named <i>sample-layout</i> when the debugger exits. You can then copy this file into your <i>.Xdefaults</i> file where it will be read when you next run the debugger. See “Customization” on page 86.
<i>-q</i>	Run <i>quietly</i> , only revealing the debugger if the program being debugged stops due to a signal or runtime exception.
<i>-v</i>	Run <i>verbosely</i> , print status information and commentary while running.
<i>-n</i>	Do not include shared object file symbols when loading the program. For large shared libraries, this option can significantly speed up the debugger and reduce the amount of virtual memory used.
<i>-p</i>	Ignore compiler-generated filename qualifiers appearing in the program symbol table. This allows source files to be found (by searching the directories specified with <i>-I</i>) even if they were moved after the executable was generated.
<i>program</i>	The name of the program to execute.
<i>args</i>	Arguments to be pass to the program.

Example

```
xcdb -I/u/derek/myproject -e2000 -c20 -i14 -i30 stuff one two three
```

invokes Xcdb and:

- Runs the program `stuff` with arguments “one two three”
- Looks for source files in either the current directory or the directory `/u/derek/myproject`
- Displays up to 2000 elements for any array
- Displays up to 20 functions in the Callers window pane
- Ignores signals 14 (SIGALRM) and 30 (SIGUSR1), passing them directly to the program without stopping

Program starting

To start a program running, left-click the Run command.

Program interrupting

To interrupt a running program and return to the debugger, point the mouse to the window from which the program was invoked and press Ctrl-c.

To resume execution, left-click the Run command.

Program terminating

To exit the debugger, left-click the Exit command.

You can also terminate the debugger and executing program by pressing Ctrl-\ on the xterm window from which you invoked the debugger. Do this only if both the debugger and the program are unresponsive to keyboard input.

Xcdb exit codes

The exit code Xcdb returns to the operating system is determined as follows:

- If the program terminated normally, Xcdb returns the value passed by the program to its `exit()` function.
- If the program terminated due to an exception, Xcdb returns 255.
- If Xcdb terminated abnormally, then a value of 1 is returned.

WINDOW ORGANIZATION

Xcdb has these windows.:

Listing	<p>Displays the source code for the function selected in the Callers or Functions windows, the file selected in the Files window, or a breakpoint selected in the Breakpoints window. The window's title indicates the file's name.</p> <p>Set or clear a breakpoint by clicking on the line to affect. If the source file was used to generate code multiple times (as for functions generated from a C++ template file or an <i>out of lined</i> inline), a menu prompts you to choose the function instance to breakpoint.</p>
Locals	<p>Displays variables defined in the scope of the function selected in the Callers window. Click on a value in this window to activates a display-format menu (see "Format Control" on page 76).</p>
NonLocals	<p>Displays variables defined outside the scope of any function (this includes static C++ class members), grouped by translation unit. Click on a value in this window to activates a display-format menu (see "Format Control" on page 76).</p>
Callers	<p>Displays a traceback of suspended function activations (most recent at top). Click on a function name to display the source code for that function in the Listing window and to display its local variables in the Locals window.</p>
Functions	<p>Displays the names of the functions comprising the program. Click on a name to display the source code for that function in the Listing window.</p>
Files	<p>Displays the names of the source files comprising the program. Click on a name to display the source code for that function in the Listing window.</p>
Breakpoints	<p>Displays a list of breakpoints currently set. Click on a breakpoint to display the source code for that breakpoint in the Listing window. Lines with breakpoints are marked with <i>stop sign</i> icons.</p>
Command	<p>Displays the commands which can be used to control the debugger. Click on command to execute it.</p>
Messages	<p>This window pane displays messages from time to time. It is invisible unless there is a message to see.</p>

WINDOW MANIPULATION

Window and mouse clicks display and control all aspects of the debugger.

Left button

The left button manipulates the *contents* of a window. To scroll a window, drag the contents; the contents scroll in a direction and amount proportional to the motion of the mouse.

Title bar	Brings up a menu:
	Move Changes the window's position
	Resize Changes the window's size
	Lower Pushes the window down
	Minimize Reduces the window to an icon
	Normalize Restores the window's original size
	Maximize Enlarge the window to fit the application window
	Horizontal S.B Togglse horizontal scrollbars on or off
	Vertical S.B. Toggles vertical scrollbars on or off
End of a scroll bar	Scrolls the contents one line or column (fast click) or one page (slow click) ¹ .
Middle of a scroll bar	Sets the window to an absolute position on the contents (position is proportional to the distance of the mouse from the end of the scrollbar).

¹ A fast click is made by pressing and releasing the button in under 1/4 second; anything else is a slow click.

Right button

The right mouse button changes the *shape, position, or visibility* of a window.

Center of window	To drag the window to a new position. Right-click without moving the mouse <i>pushes the window beneath</i> any other windows it might have been obscuring.
Corner or edge of window	To resize the window.

Keys Keys navigate through the window, and execute searches.

Enter	Makes a selection; same as left-click.
Arrow	Moves cursor, scrolling the window if necessary.
Page-up	Scrolls window back.
Page-down	Scrolls window forward.
Home	Moves cursor to first column of window.
End	Moves cursor to last column of window.
: <i>nnn</i>	Moves cursor to line number <i>nnn</i> (but not past end of file).
/ <i>XXXX</i>	Search forward to next occurrence of the string <i>XXXX</i> ; omit the <i>XXXX</i> to repeat search from current position.
\ <i>XXXX</i>	Moves cursor backward to preceding occurrence of <i>XXXX</i> ; omit the <i>XXXX</i> to repeat search from current position.

EXECUTION CONTROL

Issue commands by left-clicking on an item in the Commands window to bring up the Commands menu.

Commands	Run	Executes the program until a breakpoint is encountered or a signal is received.
	Line step	Executes the program until a breakpoint is encountered, a signal is received, or control passes to a new line of source code. Executes functions called by the current line without stopping.
	Call step	Executes the program until a breakpoint is encountered, a signal is received, control passes to a new line of source code, or a function call is made. ¹
	Return step	Executes the program until a breakpoint is encountered, a signal is received, or control returns to the caller of the current function.
	Signal	Resumes execution at the current instruction, passing whatever signal caused the interruption back to the program. Any signal sent to the program interrupts execution and returns control to the debugger. Signals can arise from: A signal key (Ctrl-c, for example) clicked in the controlling terminal's window. You probably want the program to ignore the signal and so would resume execution with the Run command. A signal received in an alarm() or wait() system call. You probably want the program to process the signal and so would resume execution with the Signal command. A signal generated by a runtime exception. Execution cannot continue, but the debugger can still inspect the environment that caused the exception. Re-execute the program with the Restart command.

Edit	<p>Invokes an editor on the file in the Listing window. Specifies the editor with <code>xcdb.Edit</code> in your <code>.Xdefaults</code>. Use <code>%s</code> and <code>%d</code> symbols for filename and line number, respectively. For example, to invoke <code>vi</code>:</p> <pre>xcdb.Edit: (xterm =+0-0 -n Vi -e vi +%d %s &)</pre> <p>To invoke <code>emacs</code>:</p> <pre>xcdb.Edit: (emacs +%d %s &)</pre> <p>To invoke <code>v</code>:</p> <pre>xcdb.Edit: (v -l %d %s &)</pre>
Restart	<p>Terminates the program, reloads it, and sets its execution point back to the beginning; all breakpoints and data format selections remain unchanged. If <code>stdin</code> is a file, it is rewound to start-of-file.</p>
Exit	<p>If the debugger was attached to a process using <code>-a</code>, then the process is allowed to resume execution (if you want the process to die, you must use <code>kill -9</code> from an <code>xterm</code> window—there's no explicit command to do this from <code>Xcdb</code>); otherwise, the process terminates and the debugger returns to the operating system.</p>
Preferences	<p>A menu prompts adjustments for <code>Xcdb</code>'s behavior. See "Preferences" on page 84.</p>

¹ Call stepping into a kernel function is not possible (because there's no way to set a breakpoint—the text segment is read only). `Xcdb` handles this by running the program until the kernel function returns to the point of call.

FORMAT CONTROL

You can reformat objects in the Locals and NonLocals windows in a variety of ways, depending on their type.

Point the cursor to an object's name or value and left-click to invoke a menu.

Point the cursor to a menu selection and click again to reformat the object as specified.

Click outside the menu (or on its title bar) to close the menu without making a change, and leave the object's format unchanged.

Common Formats

All objects share a common subset of formatting options.

Default	Displays the object's value in a representation appropriate to its type:
char	A singly quoted letter: 'a'
int	A signed integer: -123
unsigned	An unsigned integer: 4294967173
float	A floating point number: 1.23
enum	An enumerator name.
function	A function name.
class, struct, or union	A class name (or a member list, see "class, struct, and union formatting" on page 79).
array	The word "array" (or an element list, see "Array formatting" on page 80).
pointer	The word "ptr" (or a pointed-to object, see "Pointer formatting" on page 83).
Address of	Displays the object's memory address.
Type of	Displays the object's type.
Size of	Displays the object's size.
Save	Remembers the object's display format for later reference by Recall.
Recall	Changes the object's display format to match that of the object most recently referenced by Save.
Edit	Edits the object's value.

Type-specific Formats

Type-specific formatting options are also available.

Integer	Character	Letter format: 'a'
	Signed	Signed integer format: -123
	Unsigned	Unsigned integer format: 4294967173
	Octal	Octal format: 0177
	Hex	Hex format: 0x7f
Float	decimal	"f" format
	Scientific	"e" format
	Hex	Hex format: 0x7f

Complex	Decimal	Real and imaginary parts of the number in “f” format.
	Scientific	Real and imaginary parts of the number in “e” format..
	Hex	Displays the real and imaginary parts of the number in hex format
Class, Struct, or Union	Flatten	Reveals the members, horizontally.
	More detail	Reveals the members, vertically.
	Less detail	Hides the members.
Class	Show self	Runs the object’s <code>xcdb()</code> member function (if any). See “Self-displaying C++ objects” on page 85.
Array	More detail	Reveals array elements.
	Less detail	Hides array elements.
	String	Displays an <i>array of characters</i> as a null terminated string: “abc”.
	Select subrange	Selects a subrange of the array for display. A prompt asks for the subscripts of the elements you wish to see. See “Array formatting” on page 80.
Pointer	Less detail	Hides the pointed-to object.
	Hex	The pointer in hex format.
	String	A <i>pointer to character</i> as a null terminated string.
	Array	At <i>pointer to X</i> as an <i>array of X</i> .
	Select subrange	A selected subrange of the pointed-to array. A prompt asks for the elements you wish to see.
	Cast	Changes (<i>casts</i>) the base type of the pointed-to object. A list of struct, union, and typedef names prompts to select a new base type. Subsequent formatting of the pointed-to objects treats them as if they are of the type you select.
	Downcast	Converts a C++ <i>pointer to abstract base class</i> into a <i>pointer to most derived class</i> by inspecting the pointed-to object’s virtual function table pointer.
	Less detail	Hides the pointed-to object, for example:

```

class X { ... };           // base class
class Y : public X { ... }; // derived class

f() {
    X x;
    g(&x); // pass a 'pointer-to-X'

    Y y;
    g(&y); // pass a 'pointer-to-Y'
}

g(X *p) {                // at run time 'p' could be either
    //     'pointer-to-X'
    // or  'pointer-to-Y'
    //
    ...                  // click on 'p' and select 'Downcast'
    // to reveal the actual type
}

```

**class, struct, and union
formatting**

Choosing `More detail` multiple times on a structure reveals increasing levels of detail. At the minimum level of detail, only the structure name displays. At the maximum level of detail, all of the member names and values display. Similarly, clicking `Less detail` successive times causes the object's format to *fold up*. Consider the following declaration:

```
struct node
{
    struct node *next;
    struct data
    {
        int type;
        float value;
    } data;
} Node = { 0, { 1, 123 } };
```

This sequence shows how you might inspect the object:

```
Click More detail here Node: node
Click More detail here Node: { NULL data }
Click More detail here Node: next: NULL
                        data: data
Click More detail here Node: next: NULL
                        data: { 1 123.000000 }
                        Node: next: NULL
                              data: type: 1
                                    value: 123.000000
Click More detail here Node: next: NULL
                        data: type: 1
                              value: 123.000000
Click Less detail here Node: next: NULL
                        data: { 1 123.000000 }
Click Less detail here Node: next: NULL
                        data: data
Click Less detail here Node: { NULL data }
                        Node: node
```

You can also examine just a particular field of interest by clicking on that field:

[Click More detail here](#) Node: { NULL data }
[Click Type here](#) Node: { NULL { 1 123.000000 } }
[Click Hex here](#) Node: { NULL { 1 float } }
Node: { NULL { 1 0x42f60000 } }

Array formatting

Xcdb displays arrays similar to structures, except that the elements are identified by *indices* rather than *member names*. At the minimum level of detail, only the word “array” displays. At the maximum level, the indices and values of all the array elements display.

Statically allocated arrays Consider the following declaration.

```
struct point
{
    char *name;
    int coord[3];
} Set[] = {
    {"one", {1,1,1}},
    {"two", {2,2,2}},
    {"three", {3,3,3}},
    {"four", {4,4,4}},
    {"five", {5,5,5}},
    {"six", {6,6,6}},
};
```

This sequence shows how you might inspect the object:

```

Click More detail here Set: array
Click More detail here Set: { point point point point ... }
Click More detail here Set: 0: point
                        1: point
                        2: point
                        3: point
                        ...
Click More detail here Set: 0: { ptr array }
                        1: { ptr array }
                        2: { ptr array }
                        3: { ptr array }
                        ...
Click More detail here Set: 0: { ptr array }
                        1: name: ptr
                           coord: array
                        2: { ptr array }
                        3: { ptr array }
                        ...
Set: 0: { ptr array }
      1: name: ptr
         coord: { 2 2 2 }
      2: { ptr array }
      3: { ptr array }
      ...

```

Dynamically allocated arrays

In the previous section, the array dimensions were defined at compile time and known to the debugger. But for arrays with runtime defined dimensions, the debugger has no idea of the outer array dimension, so it assumes a value of 1 until you tell it otherwise. Consider the following declaration:

```

main()
{
    char **stuff = malloc(3 * sizeof(char *));
    stuff[0] = "abc";
    stuff[1] = "def";
    stuff[2] = "ghi";
    return 0;
}

```

To format `stuff` as an array of character pointers, step the program until the array has been completely initialized, and then:

```

stuff: ->->0x61
ClickString here -----
stuff: ->"abc"
Click Select subrange -----
here, enter "0,2,..."
Click More detail here -----
stuff: { "abc" "def" NULL }
stuff: 0: "abc"
       1: "def"
       2: "ghi"

```

Subrange selection

Select specific subranges of array elements by clicking on the array and choosing `Select subrange` from the menu. Then, type the subscript or range of subscripts of the element(s) that you wish to see. Use an expression of the form:

```

subrangeSpecifier ::= sectionSpecifier { ',' sectionSpecifier }...

sectionSpecifier  ::= '[' subdimensionSpecifier { ',' subdimensionSpecifier }... ']'

subdimensionSpecifier ::= lo '..' hi // subdim elements between lo and hi, inclusive
                        | lo '..' '*' // all elements of subdimension, starting at 'lo'
                        | '*' '..' hi // all elements of subdimension, ending at 'hi'
                        | '*' '..' '*' // all elements of subdimension
                        | '*' // all elements of subdimension
                        | n // n'th element of subdimension

```

The count of `subdimensionSpecifiers` must match the count of array dimensions. Here are some examples:

```

char array[4][2]; // a 4 by 2 array

[0, *] // matches elements: [0,0]
                                     [0,1]

[1..2, 1], [ 3, 0..1] // matches elements: [1,1]
                                                         [2,1]
                                                         [3,0]
                                                         [3,1]

```

If a subrange specifier would display more than 1,000 elements, then the remainder display as "...". Change this limit by specifying a different value using `-e` or the `xcdb.ArrayLimits` item in `.Xdefaults`.

Pointer formatting

At the minimum level of detail, only the word “ptr” displays for a pointer object. Click More detail to reveal the pointed-to object. Consider the following:

```
typedef int (*FUNCP)();          /* a function pointer */
FUNCP Table[3] = { main, exit }; /* table of pointers */
```

The sequence below shows how you might inspect the object:

```
Click More detail here Table: array
Click More detail here Table: { ptr ptr NULL }
Click More detail here Table: 0: ptr
                        1: ptr
                        2: NULL
Click Type here Table: 0: -> main()
                        1: ptr
                        2: NULL
Click Type here Table: 0: -> function-returning-int
                        1: ptr
                        2: NULL
Click Type here Table: 0: pointer-to-function-returning-int
                        1: ptr
                        2: NULL
Table: 3-item-array-of-pointer-to-function-returning-int
```

BREAKPOINTS

Set or remove unconditional breakpoints by clicking on the line in the Listing window. Set or remove conditional breakpoints that relate to the line indicated by the *pointing hand* icon as follows:

- 1 Run the program to the line where the breakpoint is to be set.
 - Ⓐ If you set a breakpoint to get there, remove it.
- 2 Left-click on an integer or pointer object in the Locals or NonLocals window, and select Breakpoint from the menu.
- 3 Enter a *breakpoint trigger value* for the object, at the prompt.

Xcdb indicates the breakpoint with a *stop sign* icon on the source line and with an asterisk-marked (*) entry in the Breakpoints window

Xcdb stops the program whenever the specified line executes, and the object has the specified trigger value.

PREFERENCES

To specify your preferences, use the Preferences option from the Commands menu in the Commands window.

Preference settings	Language	Controls printing of variable names and interpretation of array element addresses. Normally, Xcdb determines the language automatically, based on the initial stopping point in the program. You can change this by clicking either mouse button to cycle through the possibilities:
	C	Array element addresses are computed in <i>row major</i> form.
	C++	Array element addresses are computed in <i>row major</i> form; variable names are <i>demangled</i> ; nested class members are labeled.
	FORTRAN	Array element addresses are computed in <i>column major</i> form.
	Variables	Controls printing of variables in the Locals window pane.
	Lexically scoped	Displays only the variables in the scope of the current instruction.
	Unscoped	Displays all variables in the current function, even those in other lexical blocks. This option is a work-around for a bug in some compilers—see “Frequently asked questions” on page 88.
	Secret variables	Controls visibility of C++ compiler-generated variables.
	Hidden	Does not display secret variables.
	Visible	Displays secret variables.
	Include Files	Controls interpretation of file symbols appearing in the symbol table.
	Respect	The debugger makes use of <code>#include</code> file information appearing in the symbol table.
	Ignore	The debugger ignores <code>#include</code> file information appearing in the symbol table. This option is a work-around for bugs in <code>cpp</code> , <code>cc</code> , and <code>cfront</code> —see “Frequently asked questions” on page 88.
	File search path	Specifies the directories to search when displaying source files in the Listing window. Enter a list of directory names, separated by spaces. See also the description of <code>-s</code> .
	Upon fork follow	Controls tracing of <code>fork()</code> system calls:
	Parent	Follows the parent process after a <code>fork()</code>
	Child	Follows the child process after a <code>fork()</code>
		When stepping through a <code>fork()</code> statement, you must use Line Step and not Call Step; otherwise, the debugger gets stuck trying to trace the system call.
	Autoraise	Controls automatic raising of interior window upon mouse entry.
	Detail per click	Controls the count of levels of detail to reveal (hide) when requesting More detail (Less detail) on a structure, union, array, or pointer object. Right-click to increase the value, and left-click to decrease it.

SELF-DISPLAYING C++ OBJECTS

This is an experimental feature that allows C++ objects in a program to *show themselves* in response to a request from the debugger. When a C++ object is selected on the Locals or NonLocals window, and you choose Show self from the menu, Xcdb executes a member function named `xcdb()`, if found. For every class you wish to examine, write an `xcdb()` member function with these constraints:

- ※ no arguments
- ※ of type `void`
- ※ must *not* be inline
- ※ every class must have its own `xcdb()` member function (they cannot be inherited; they may be virtual, but must be defined for each subclass)

When you want a class instance to run its `xcdb()` member function, click on the object (as usual), format the object as a “structure” (choose More Detail if you only have a pointer to the object), and choose Show self. This runs the object’s `xcdb()` member function. Control then returns to the debugger.

An `xcdb()` member function can be written to do anything at all. It might say something interesting, display pretty pictures, and so on. Use your imagination.

Example

```
class Mumble
{
private:   const char *name;
public:   Mumble(const char *name) : name(name) {}
public:   const char *name() { return name; }
public:   void xcdb();
};

void Mumble::xcdb() { printf("My name is '%s'.\n", name()); }

main()
{
    Mumble& mumble = *new Mumble("mumble");
}
```

Clicking on the variable “mumble” in the Locals pane and selecting Show self from the menu displays

```
My name is 'mumble'.
```

in the xterm window that invoked the debugger.

Notes

Attempting to Show self on a class or struct for which no `xcdb()` member function is defined produces a complaint, but is otherwise harmless.

Any breakpoint or exception inside the `xcdb()` member function, while running in the context of Show self, terminates the function (returning control to Xcdb), and is otherwise ignored.

CUSTOMIZATION

Change `xcdb`'s window shape, position, font, colors, and window layouts with `\$HOME/.xcdbdefaults`. For information about available fonts and colors see `/usr/lpp/X11/defaults/Xfonts` and `/usr/lib/X11/rgb.txt`, respectively.

The following tables summarize the `.xcdbdefaults` entries. Values to the right of the colon indicate acceptable entries, where:

geometry is a geometry specification such as "100x300+10-5"
font is the name of a font, such as "Rom10.500"
color is the name of a color, such as "Slate Blue" or "\#7AD"

General

Geometry: <i>geometry</i>	Main window size and placement
Font: <i>font</i>	Font to use for text
AutoRaise: on off	Behavior of window when mouse enters
SaveUnder: on off	Handling of pixels obscured by popup menus. On some X servers, popup menus run faster with SaveUnder set on; others run faster with SaveUnder set off. Try both settings and see which works best for you.

Layout

The layout entries customize each window in the debugger. You must specify settings for all or none of the windows; you cannot specify some of the windows.

Create layout entries from your working environment with `-l`; see "Running" on page 70 for information.

SpecialLayout: yes no	Do window specifications follow?
<i>xxxx</i> Geometry: <i>geometry</i>	Size and placement for normal window
<i>xxxx</i> IconGeometry: <i>geometry</i>	Size and placement for iconized window
<i>xxxx</i> IconifyOk: yes no	Permit iconization of this window?
<i>xxxx</i> IconStartup: yes no	Iconize window at start-up?
<i>xxxx</i> Scrollbars: vertical horizontal both none	Scrollbar style

where *xxxx* is one of Callers, Functions, Files, Breakpoints, Commands, Listing, Locals, NonLocals, Formats, or Messages.

Color	BorderIdle: <i>color</i>	Window borders, mouse outside
	BorderActive: <i>color</i>	Window borders, mouse inside
	Foreground: <i>color</i>	Normal text
	Background: <i>color</i>	Normal text
	MouseBody: <i>color</i>	Mouse body
	MouseOutline: <i>color</i>	Mouse outline
	CursorForeground: <i>color</i>	Cursor
	CursorBackground: <i>color</i>	Cursor
	MarkForeground: <i>color</i>	Marked text
	MarkBackground: <i>color</i>	Marked text
	TitleForeground: <i>color</i>	Window pane titles
	TitleBackground: <i>color</i>	Window pane titles
	DialogForeground: <i>color</i>	Command lines
	DialogBackground: <i>color</i>	Command lines
	DimForeground: <i>color</i>	Non-selectable menu items
	DimBackground: <i>color</i>	Non-selectable menu items
	ScrollbarIdle <i>color</i>	Scroll buttons, mouse outside
	ScrollbarActive <i>color</i>	Scroll buttons, mouse inside
Other	Editor: <i>command</i>	The specified <i>command</i> is invoked when the Edit command is selected from the Commands window (see earlier).
	Language: <i>language</i>	The debugger's behavior is adjusted for the specified <i>language</i> , as described in the Preferences menu section (see earlier). <i>language</i> must be one of: <ul style="list-style-type: none"> ❖ C ❖ C++ ❖ FORTRAN
	RespectIncludeFiles: yes no	Controls interpretation of file symbols appearing in the symbol table, as described in the { <code>\it Preferences</code> } menu section (see earlier).
	ArrayLimits: <i>NNNN</i>	Controls data formatting, as described for the “-e” command line flag (see earlier).
	DetailPerClick: <i>NNNN</i>	Controls data formatting, as described for the “-d” command line flag (see earlier).
	UnsignedCharFormat: decimal hex	Selects default data formatting style for unsigned char numbers.
	UnsignedShortFormat: decimal hex	Selects default data formatting style for unsigned short numbers.

Example

```

xcdb.Font: Rom17.500
xcdb.Background: slate blue
xcdb.Edit: (emacs '+%d' '%s' &)
xcdb.RespectIncludeFiles: yes
xcdb.ArrayLimits: 2000
xcdb.DetailPerClick: 2
xcdb.UnsignedCharFormat: hex
xcdb.FloatFormat: scientific
xcdb.AutoRaise: on
xcdb.SaveUnder: off

```

FREQUENTLY ASKED QUESTIONS

Here are the answers to some frequently asked questions.

Q: This document makes reference to menu item XXXX, but I don't see it on my menu.

A: Your window pane is either too small or the item has scrolled out of view. Press Home and then use the cursor keys to scroll the window contents until you find the item you are looking for.

Q: A window pane or menu appears to be empty.

A: See the answer to the previous question.

Q: My program runs fine when invoked from the debugger, but doesn't run when invoked from the shell command line.

A: Unlike the command shell, Xcdb loads your program without searching the \$PATH environment variable. You've probably got a program by the same name somewhere in your \$PATH. Try explicitly qualifying the program name when you type it on the command line. For example, type:

```
./test a b c # run program in current directory \end{verbatim}
```

instead of:

```
test a b c # oops, this probably invokes /bin/test \end{verbatim}
```

Q: The debugger stops with a Signal 0 when it encounters the `system()` function in my program.

A: This is normal. Just click the Signal item on the command pane to continue, or reinvoke Xcdb with “-i 0.”

Q: I can't set a breakpoint on some lines of my C++ program (compiled with `cfront`).

A: There are bugs in `/lib/cpp`, the preprocessor used by `cfront` to perform macro expansion. Try another macro preprocessor—some people have had luck with `/usr/lpp/X11/Xamples/util/cpp/cpp`. Point to it with the `CC`'s “`cppC`” environment variable, and then recompile.

There are also bugs in `cfront` related to generation of `#line` directives for templates and include files. Try setting Include files: *Ignore* in the Preferences menu and see if this helps.

Q: Xcdb displays the wrong source file and/or line number in my C++ program (compiled with `cfront`).

A: Try setting Include files: *Ignore* in the Preferences menu and see if this helps.

Q: Xcdb displays the wrong source file and/or line number in my C++ program (compiled with `x1C`).

A: Make sure you have set Include files: *Respect* in the Preferences menu. Another possibility is that the source file contains more than 65,534 lines. Due to an AIX symbol table design *feature*, line information for such files is stored incorrectly. The only workaround is to split the source file into smaller pieces.

Q: I can't see one of my local variables, but I know it's there.

A: This is due to a compiler bug. Try the Variables: *Unscoped* option on the Preferences menu.

Q: My program seems to be running correctly, but the variables displayed by Xcdb look wrong.

A: You probably compiled your program with both `-g` and `-O`. The resulting compiler optimizations confuse the debugger. Recompile your program with either `-g` or `-O`, but not both.

Q: I can't see code generated from `#include` files.

A: You need a newer version of x1C (such as version 01.02.0000.0000, or later).

Q: Xcdb complains about an *ambiguous breakpoint* when I try to set a breakpoint on certain parts of my program.

A: You probably tried to set a breakpoint on an instruction that was one of several "instantiations" generated from the same `#include` file.

If you are debugging template code generated by the x1C compiler, make sure you've set the Language: C++ option on the Preferences menu.

Otherwise, if you are debugging non-template code, or code generated by compilers other than x1C, there is no mechanism by which xcdb can infer the instruction instantiation to which you refer, so it is not possible to set a breakpoint on the specified line. Sorry.

Q: I can't see a traceback in the Callers window pane when I set a breakpoint in a signal handler.

A: This is a deficiency in xcdb that is being addressed.

Q: I get an error when attempting to attach the debugger to a process using `-a`.

A: This seems to have something to do with shared libraries. If you can reproduce this problem with a small program, please send a bug report to the Taligent Tools Team.

Q: Xcdb is sluggish when stepping. How can I make it faster?

A: Display update performance during stepping operations can be improved by *iconifying* the NonLocals window pane if it is not needed. The debugger is then saved the expense of reading and formatting (potentially large) amounts of global data from the program's execution image. Also, choosing the `-n` command line option will help here, by reducing the number of symbols that Xcdb must search. Reducing the size of the main window or using a larger font will also help, because it reduces the amount of window drawing that takes place. Also, enabling `xcdb.SaveUnder` in your `.xdefaults` file may improve performance of pop-up menus (see "Customization" on page 86).

Q: How can I format a number of variables, all in the same style, without tediously clicking *more detail* on each one?

A: Try using the Save and Recall selections on the Formats menu to propagate the formatting information from one object to all the others.

Q: How can I change the display format of all the elements of an array at once, without tediously clicking on each one?

A: Try this:

- ❶ Format the first item in the array
 - ❷ Use *Select subrange* to (re)select the elements you wish to see
The format of the first element propagates through to all the other elements
-

Q: How can I invoke Xcdb from inside my program?

A: Try something like this:

```
main()
{
    foo();
}

foo()
{
    bar();
}

bar()
{
    trouble();
}

trouble()
{
    extern char **p_xargv; /* undocumented variable */
    char cmd[100];
    sprintf(cmd, "xcdb -a %d %s", getpid(), p_xargv[0]);

    if (fork() == 0)
        system(cmd);          /* runs Xcdb */
    else
        pause();              /* waits until Xcdb issues "Run", "Line Step", etc. */
}
```

Q: When I *Select a subrange*, I only see the first 1,000 elements of my selection. Where are the rest?

A: As a safety feature, `xcdb` displays at most 1,000 elements per array. Use `-e` or `xcdb.ArrayLimits` in your `.xcdb.defaults` file to change this limit.

Q: How can I display a region of memory as an unstructured hex *dump*?

A: Try this (ok, it's a bit of a kludge, but it works):

- 1** Determine the address of the region you wish to inspect (using `Format...as address`, for example)
- 2** Take any convenient char pointer in your program and set its value to the address you wish to inspect (using `Edit`)
- 3** Select the number of elements to be displayed (using `Select subrange`)

Q: What version of `xcdb` do I have?

A: Type `xcdb` (no arguments) to find out.

Q: Where can I get the latest version of `xcdb`?

A: Obtain XCDB6000 PACKAGE from your nearest AIXTOOLS service machine.

Q: What's new in the latest version of `xcdb`?

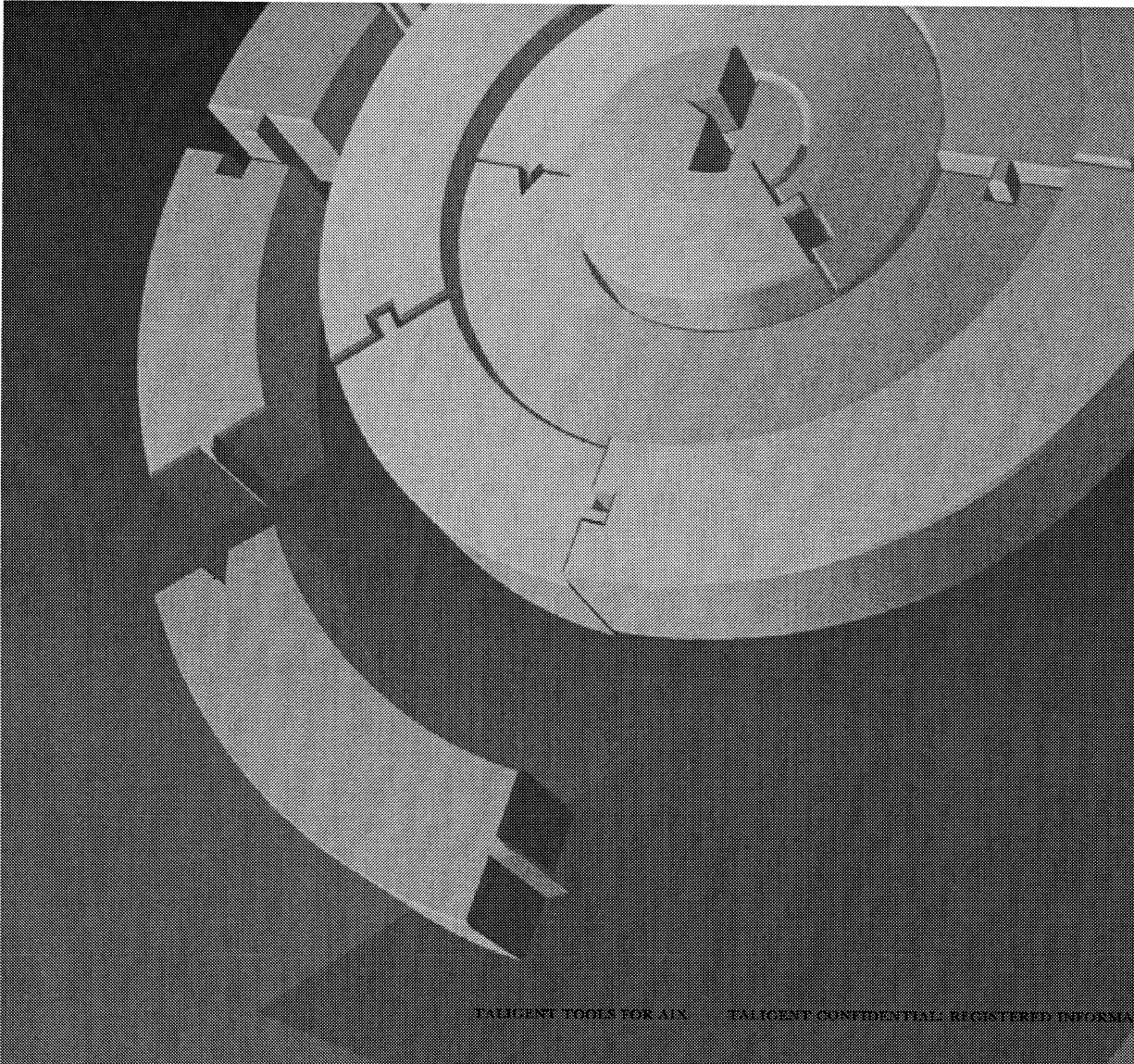
A: Please read the XCDB6000 NEWS file that is shipped with each XCDB6000 PACKAGE.

Q: I have a question that isn't answered here.

A: Please report any problems you discover (or wish list items) to the Taligent Tools Team.

REPORTING BUGS

If you encounter a problem with Xcdb, file a Taligent bug report.



APPENDIX A

TIPS & TECHNIQUES

Everybody has their own work style, but there are some simple tricks you can do to make yourself more productive. Here are some useful pointers.

CDPATH

The `cdpath` shell variable contains a list of directories that shell searches when you use `cd`. For example, if you are in your `$HOME` directory, you can type:

```
cd Envious
```

and the shell will take you right there. The shell looks in the current directory first, and if it does not find `Envious` there, it searches the directories in `cdpath`, which is what happens in the previous example.

This little trick saves a massive amount of typing when you are navigating around the Taligent source tree. Here is an example of settings to add to your `.cshrc` file:

```
set cdpath=( ~ \
    ${HOME} \
    ${HOME}/Taligent \
    ${HOME}/tools \
    ${HOME}/Taligent/Toolbox \
    ${HOME}/Taligent/Toolbox/InternationalUtilities \
    ${HOME}/Taligent/Toolbox/Document2 \
    ${HOME}/Taligent/Toolbox/Runtime \
    ${HOME}/Taligent/Albert/Main \
    ${HOME}/Taligent/Instrumentation/TestSystem\
    ${HOME}/Taligent/Time \
    /home/local \
)
```


XCDB—THE DEBUGGER

Taligent uses `xcdb` (an internal IBM project) as its Taligent Application Environment debugger. Be sure to read Chapter 6, “Xcdb,” before using this debugger. To make your work with `xcdb` easier, use the suggested `.Xdefaults` file for standard behavior.

Instead of calling `xcdb` directly, use the `xdb` script which install `SCMFetch` and turns off some interrupts that you probably do not need.

OPUSBUG()

Within the Taligent Application Environment, `OpusBug()` is a function that calls a UNIX program script which runs a debugger to attach to your running process. `OpusBug()` emulates the functionality of the `DebugStr()` call found in many 68K development environments. While fairly limited because the UNIX environment is very different than other development environments, `OpusBug()` provides the rudiments of printing a message and starting a debugger.

 **NOTE** The origin of the name *OpusBug* is lost in obscurity.

When you call `OpusBug()` within the Taligent Application Environment, it

- prints a message.
- uses `system()` to call `pink_debugger`: the program script. `pink_debugger` must be in your `$PATH`.
- then puts your process to sleep for five seconds. This is generally enough time for a debugger to get started and attach to the process to be debugged. The debugger comes up with `sleep()` on the top of the stack; below `sleep()` should be `OpusBug()` and then the routine that called `OpusBug()`. You should be able to debug from there.

Because `OpusBug()` invokes `pink_debugger` via a `system()` call, it carries a few restrictions:

- The `pink_debugger` script must terminate with an exit status of zero.
- The `pink_debugger` script must not be blocking. This means that anything that requires interaction, like a debugger, must be run in the background.

Here is the prototype for `OpusBug()`:

```
void OpusBug(char *message); // Print the message, and call pink_debugger
```

`OpusBug()` passes two arguments, the process ID and the calling program name, to provide enough information for a debugger to attach to a running process.

Here is a sample `pink_debugger`.

```
#!/bin/sh
#
# This program starts an xdb session in the background.
#
```

Arguments from
OpusBug()

```
PROCESS_ID=$1
PROGRAM_NAME=$2
```

Call the debugger

```
echo
echo
echo "*** Entering pink_debugger ***"
echo "*** PROCESS_ID == $PROCESS_ID ***"
echo "*** PROGRAM_NAME == $PROGRAM_NAME ***"
taldb -a $PROCESS_ID $PROGRAM_NAME &
echo "*** Exiting pink_debugger ***"
exit 0
```

Must return 0

To print the message, but not start a debugger, pink_debugger should be nothing more than exit with a zero return status.

```
# Do not start the debugger
exit 0
```

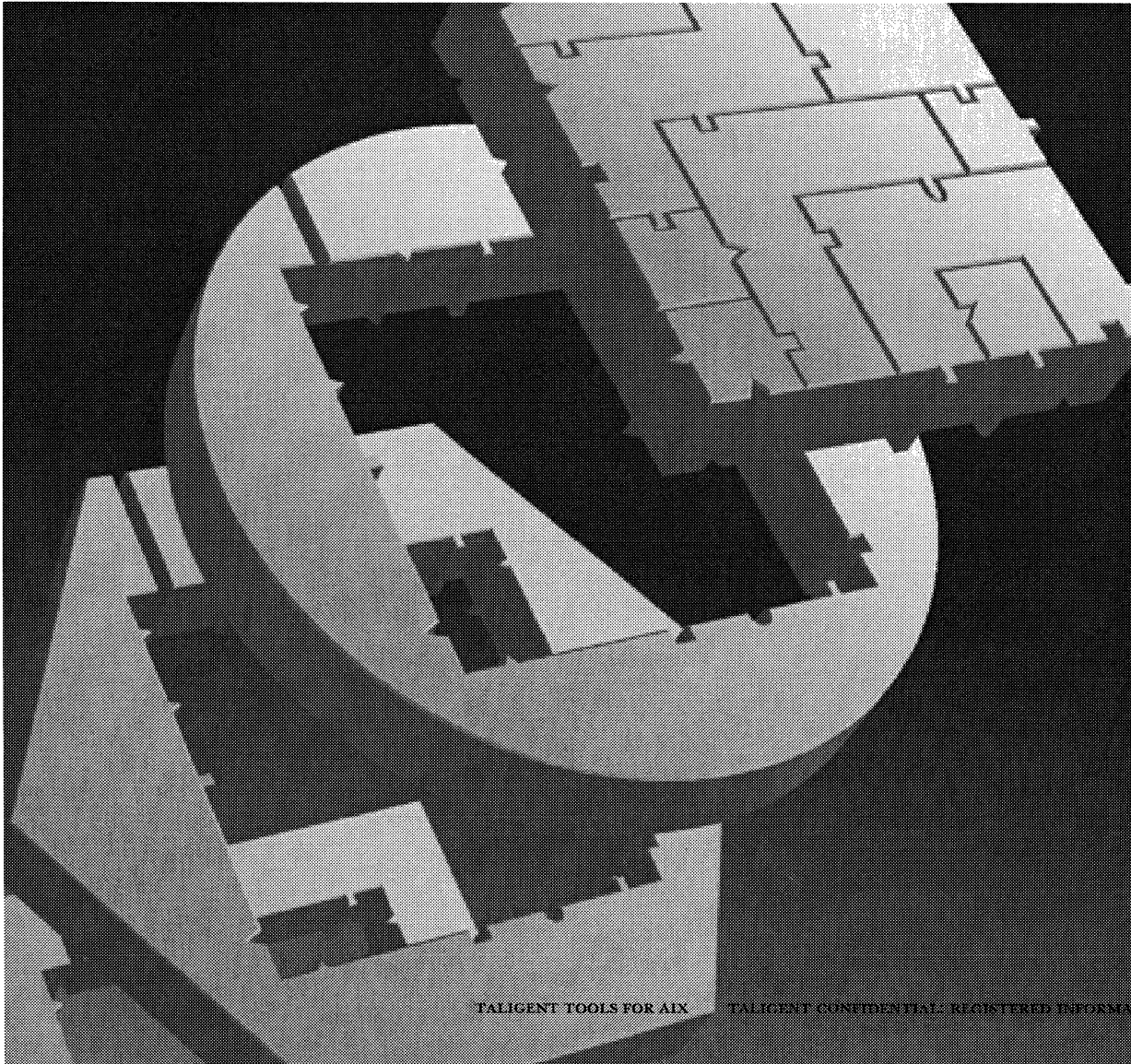
To neither print a message nor start a debugger (do nothing), set the PINK_DONT_USE_OPUSBUG environment variable.

```
setenv PINK_DONT_USE_OPUSBUG
```

PART 2

TEST ENVIRONMENT

Chapter 7		
Test Environment Overview		101
Chapter 8		
Test framework		103
Chapter 9		
RunTest		117



CHAPTER 7

TEST ENVIRONMENT OVERVIEW

The Test environment provides the tools and protocols for developing tests to ensure that your code works properly. The Test environment gives you a standard way to connect your code to the test conditions and get the test results.

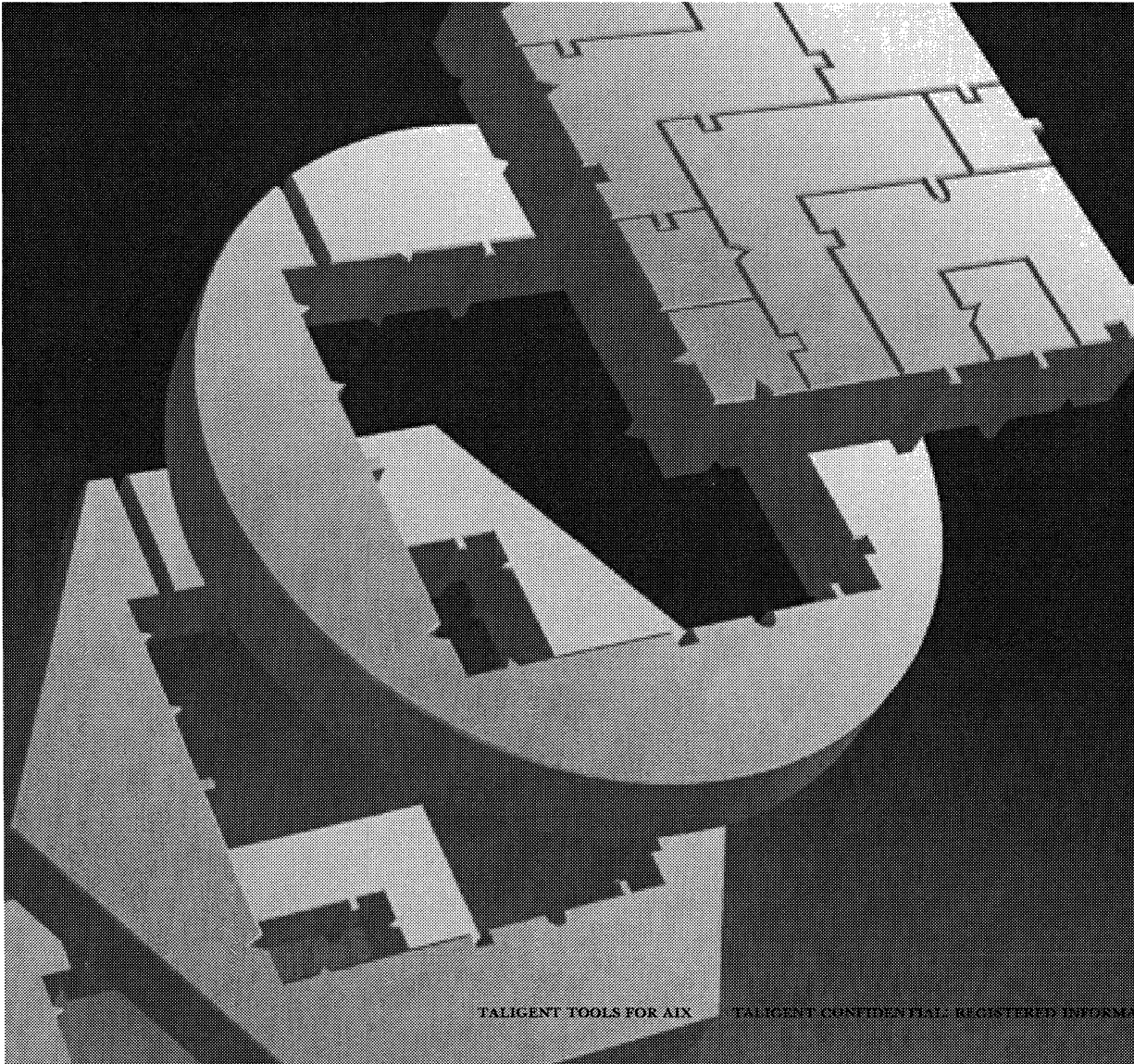
The Test environment consists of two major elements.

The Test framework is a collection of classes that provides a standard format for all tests and results reporting. The design goal is that anyone can:

- ✦ Run any test
- ✦ Understand the results of any test

The Test framework also eliminates the need to reinvent solutions to recurring problems. For example, the Test framework contains a ready-to-use class to test classes derived from MCollectible for proper support of flattening and cloning.

The RunTest application gives you an execution environment that works with a scripting language that allows you to run tests, examine them at run time, and retrieve logged outputs after run time.



CHAPTER 8

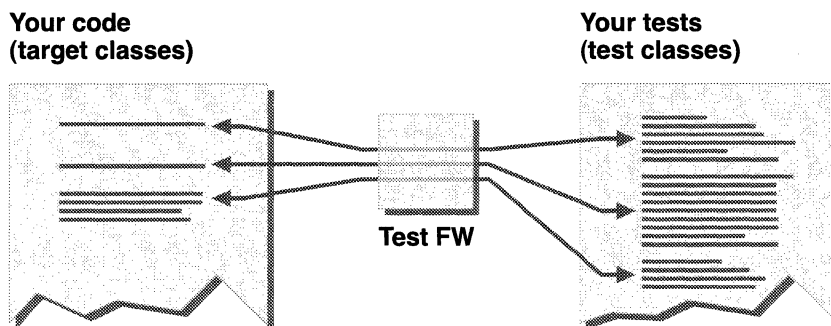
TEST FRAMEWORK

The Taligent Operating Environment Test framework provides the structure for you to link the tests you write to your code. You can then perform any Test-framework compliant tests consistently with the RunTest application.

In addition to performing a test, the Test framework supports:

- ✦ Setup and cleanup operations for each test
- ✦ Different ways to combine tests
- ✦ Test logging and timing

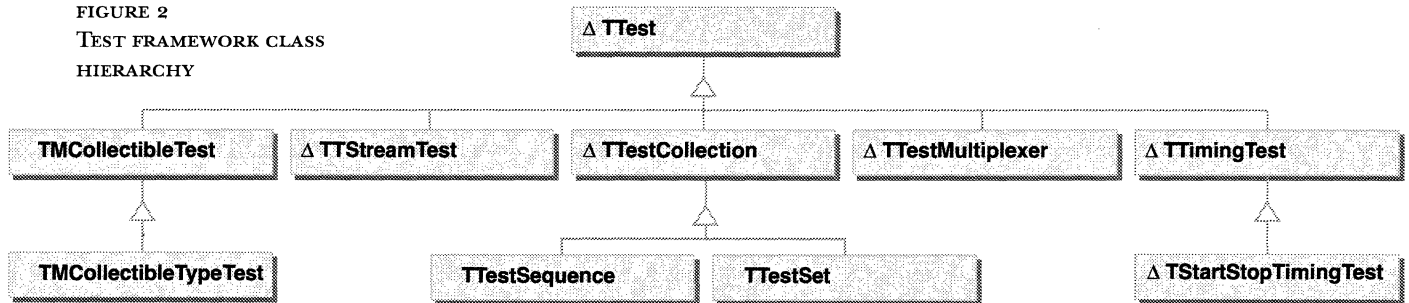
FIGURE 1
THE TEST
FRAMEWORK
PROVIDES A
CONSISTENT
INTERFACE
BETWEEN YOUR
CODE AND YOUR
TESTS



TEST FRAMEWORK OVERVIEW

The core of the Test framework is the abstract base class TTest. Your test is a derived class of TTest or one of its derived classes.

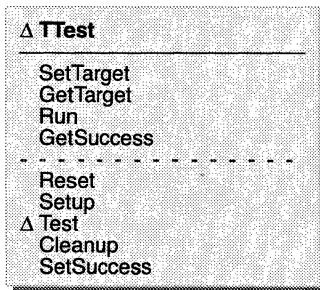
FIGURE 2
TEST FRAMEWORK CLASS
HIERARCHY



TTest Class

TTest is an abstract base class. Use this class to decide whether a well-defined test target works. Create a derived class of TTest for tests that perform a single function. TTest contains a Test member function that you must override with the code that represents your decision function. A decision function is code you write that returns a True or False result when you test a specific condition.

FIGURE 3
IMPORTANT
TTEST
MEMBER
FUNCTIONS



When you declare a derived class of TTest to be a friend of the target class, the derived TTest class can access all the private interfaces of the target class for internal testing purposes.

Test collection classes

The Test framework contains derived classes of `TTest` that allow you to group tests that look at different aspects of your code.

TTestCollection, an abstract base class of `TTest`, contains a collection of `TTest` instances. The tests in the collection run sequentially to determine the success or failure of the entire group. `TTestCollection` has two concrete derived classes:

- ✦ `TTestSet` contains an unordered set of subtests that can be shuffled to vary the order the subtests run.
- ✦ `TTestSequence` runs its subtests in a fixed sequence.

TTestMultiplexer derives from `TTest` and supports multiple decision functions applied to a single test target. Invoke these decision functions by using text keys. You can write a group of decision methods, then build a table that maps keys to methods.

Protocol tests

Protocol tests allow you to test an entire tree of classes if those classes are all expected to adhere to a protocol. The Test framework contains two protocol tests, one for derived classes of `MCollectible` and another for classes not derived from `MCollectible`.

TMCollectibleTest tests the implementation of the `IsEqual`, `Hash`, `Clone`, `operator>>=`, and `operator<<=` members of classes derived from `MCollectible`. You do not need to derive or customize `TMCollectibleTest`, it is immediately ready for use by anyone writing a derived class of `MCollectible`.

You can also change or augment `TMCollectibleTest` to test an enhanced protocol superset by overriding some of its members.

`TMCollectibleTypeTest` is a derived class of `TMCollectibleTest`, which is currently defined by macros. It tests certain behaviors that cannot be tested without a template class including the `operator ==`, the assignment operator, the copy constructor, and the constructor and destructor.

TStreamTest is an abstract base class that tests the implementation of `operator>>=` and `operator<<=` for classes *not* derived from `MCollectible`. Unlike `TMCollectibleTest`, you derive from `TStreamTest` for each target class you want to test. Use the declaration and definition macros supplied with `TStreamTest` to create the required derived classes.

Timing tests

The **TTimingTest** base class provides a basic guide for tests that measure the time a specific operation takes to complete. `TTimingTest` contains three framework members: `TimingSetup`, `TimingTest`, and `TimingCleanup`.

Only the `TimingTest` member function is timed. The `TimingSetup` and `TimingCleanup` members are run before and after `TimingTest` but are not timed themselves. `TTimingTest` produces statistical analysis of results.

Related classes

The Test framework uses other classes not directly related to TTest that are useful to derived classes of TTest.

TTieredText is the class of objects collected by TTieredTextBuffer. It is a derived class of TText and allows a Test framework user to assign a tier of detail to each instance. Tiers are, in increasing level of detail: headline, general, normal, detail, and debug.

TTieredTextBuffer behaves like the C++ ostream class. It contains <<operators for all basic types. Unlike the ostream class, TTieredTextBuffer keeps a collection of all text sent to it. Other features of TTieredText are echoing of text to a destination you specify, filtering output so that detailed information is suppressed or displayed, and flushing text beyond a certain level of detail from the buffer. Each instance of TTest contains a TTieredTextBuffer to which derived classes can stream diagnostic text messages. TTest itself uses this mechanism to report progress and results.

TextArgumentDictionary parses a sequence of TText objects into pairs of keys and values. This allows you to check quickly for the existence of a keyword on the command line or to retrieve the value given for a certain option.

GETTING STARTED WITH THE TEST FRAMEWORK

Here is an example of how you create a simple test of a member function.

Assume you have a class named TMySample that has a member function, Add, which adds two objects together. You want to know if TMySample::Add works correctly.

You need to decide the conditions you want to test and what the proper result should be. For example, you want to make sure that your Add function properly handles integers and complex numbers (a type you have defined). You would write a test member that has an operation that adds an integer and a complex value and compare that result with the result you expect. If the result is what you expect, your test succeeds, otherwise, your test fails.

The following steps summarize the way you would create a test with the Test framework:

- 1 Create a new derived class of TTest, TMySampleTestAdd.
This derived class contains the decision condition for the Add function.
- 2 Verify that an instance of TMySample is available and in the proper state for the test.

The required TMySample instance is called the *target* and a pointer to it is maintained using TTest::SetTarget and TTest::GetTarget.

To determine the correct state, you might call Get functions in the target.

3 Override the Setup member function.

Parse input arguments using `TTextArgumentDictionary`, if needed, and initialize any private data members used for the test.

If `TMySampleTestAdd` is part of a `TTestCollection`, it might share its target with the other subtests in the group.

See “Setting Up the Environment” later in this section.

4 Override the Test member function.

Put the code that determines whether the Add member works correctly here. Decide if the test has passed or not and call `SetSuccess`.

Example code for test here

See “Creating a Test” later in this section.

5 Override the Cleanup member function

Perform any cleanup after performing the tests. See “Cleaning Up After a Test” later in this section.

6 Compile and link into a shared library.

When you are ready to perform the test, use the `RunTest` application.

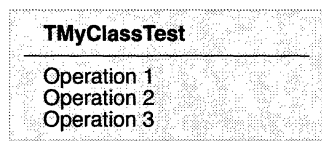
DESIGNING A TEST

A single `TTest` covers an area, large or small, which allows you to determine what works and what does not work when the test succeeds or fails. A single `TTest` can exercise a single member function or can parse the input to select a subset of member functions.

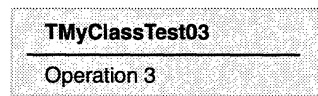
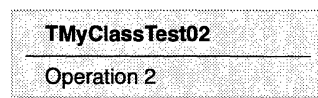
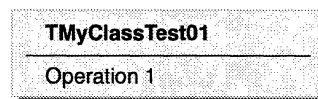
You decide how much testing a single `TTest` derived class needs to perform. If a single `TTest` turns up several defects, the scope of your test is too large.

The most important part of using the Test framework is designing a test or group of tests that properly exercises your code.

FIGURE 4
EACH OF
YOUR TEST
CLASSES CAN
PERFORM
MULTIPLE
OPERATIONS



`TMyClassTest`
performs
several
operations on
`TMyClass`



Each of the
`TMyClassTestXX` classes
perform a single operation

Before you use the Test framework to organize your tests, decide what operations you can use for each class to test the public, protected, and private interfaces.

For example, you might find that for a given instance of `TMyClass`, which contains an `Add` function, you can perform several different operations to test that instance. Each operation might take arguments. You might want to have several tests of the `Add` function, passing the test different combinations of arguments that the `Add` function is designed to process, such as reals or vectors.

Cast each operation as a decision: Does the operation work correctly? The outcome of the test condition must be `True` or `False`, indicating the success of the associated test.

As a special case, when you discover that some sequence of actions causes a defect, you need to write a special test that causes that sequence of actions to occur in order to implement regression testing.

CREATING A TEST

The minimum requirement to use the Test framework is to include `Test.h` in your source and to link your shared libraries with `TestFrameworkLib`.

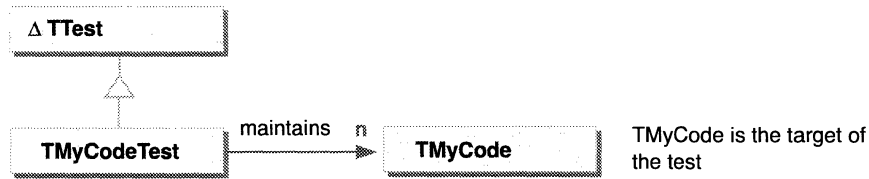
To use other features of the Test framework, use the following files:

```
TestCollection.h
TestMultiplexer.h
MCollectible.h
StartStopTimingTest.h
StreamTest.h
TimingTest.h
MCollectibleTest.h
TieredTextBuffer.h
TextArgumentDictionary.h
TieredText.h
```

Your test class and the class you are testing, the target class, have no inheritance relationship. Specifically, the test class is not a derived class of the target class. Instead, each test class has an instance of its target class.

Use `TTest::SetTarget` and `TTest::GetTarget` to set and get a pointer to some instance that a `TTest` is testing.

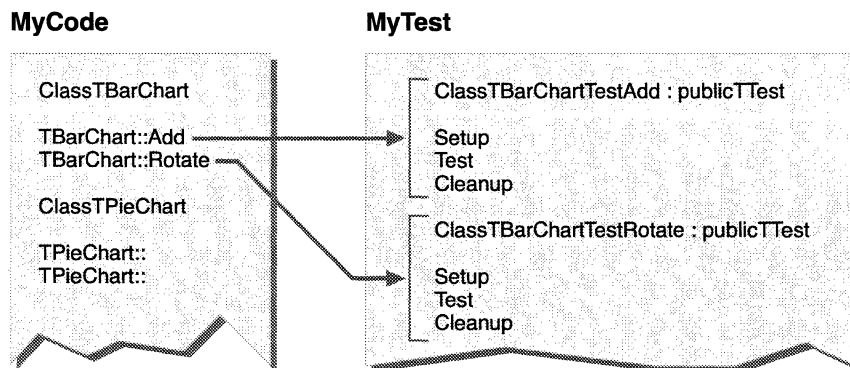
FIGURE 5
TMyCodeTest
CONTAINS THE
CODE FOR YOUR
TEST IN THIS
EXAMPLE



The TTest instance that owns the target needs to delete the target when the TTest is destroyed. The owning TTest must cast the void* target to its correct type, or at least the correct base type, before deleting the target, or else the correct destructors will not get called.

You can use a simple naming convention to clarify which test classes relate to the classes you are testing. For example, the test class for TMyClass is called TMyClassTest. The test class for TMyClass::MyMethod is TMyClassMyMethodTest.

FIGURE 6
EACH MEMBER
FUNCTION
THAT YOU
WANT TO TEST
NEEDS AN
INSTANCE OF A
TTEST OBJECT



Writing a test function

When you create your test class, you override the Test member function. The Test function contains the operations you use to actually perform the test.

Based on the result of your test operation, your Test function must call SetSuccess. Call SetSuccess with True if it passed. Call SetSuccess with False if it failed.

Setting up the environment

TTest::Setup performs arbitrary setup in preparation for the Test member function. Override Setup to specify any conditions that the test requires.

Run calls Setup before calling Test.

If a test is unable to run more than once after the first test runs, have the Setup function cause the test to fail on subsequent runs with an appropriate message.

Cleaning up after a test

TTest::Cleanup is executed even if a software exception occurs. Override Cleanup to perform any actions after the test finishes.

Run catches exceptions that occur in Test and always calls Cleanup after calling Test.

Note that if a hardware exception occurs in Test, such as a bus error, then Cleanup is not called. Such a hardware exception will probably kill the thread that ran the test but not necessarily the task in which the test was running.

Writing a test to run more than once

To run a test more than once, override TTest::Reset to change the state of the test so that you can run the test again. Reset is a public member that any test class can call. It is also called automatically inside Run if a test has been run and has not been reset.

To specify the number of times to run the test, use the -n option of the RunTest application.

Overriding inherited MCollectible members of TTest

Derived classes of TTest need to override the streaming operators when member variables are added to the class. These new member variables must also be streamed for the test to be functional.

Override Hash and IsEqual when you want additional derived class information to be considered for hashing and equality testing.

Writing text to the console

To write out diagnostic text from a test, use the member function `OutputTextStream`, which understands the standard C++ operator `<<` for all built-in types. Text output produced with this member goes to the console and is also saved in a `TTieredTextBuffer` within the test. You can then log the test, including the text buffer.

If a test fails, you can retrieve the diagnostic text associated with the test to try to determine the cause of the failure.

`TTieredTextBuffer` sends output to the console. Text that is output in this way is saved with the test. This is useful for evaluating the test after resurrecting it from a log. However, saving all text to a log can sometimes cause a problem if it uses too much memory.

```
void TMyClassTest::Test() // This member is in a derived class of TTest
{
    OutputTextStream() << "Hello, world\n";
}
```

To cause some messages to appear on the console but not be saved, use the special tier `kEphemeral`. Note that this is just a special case of the usual tier mechanism:

```
void TMyClassTest::Test() // This is a derived class of TTest
{
    // This will be recorded in the test
    OutputTextStream() << "Important data = " << fData << '\n';

    // Subsequent text will NOT be recorded in the test, to save memory
    OutputTextStream() << PushTier(TTieredText::kEphemeral);
    for (short i=0; i<32000; ++i) {
        OutputTextStream() << "Loop " << i << '\n';
        ...
    }

    // Start recording text again
    OutputTextStream() << PopTier() << "New data = " << fData << '\n';
}
```

You may want to use a `TTieredTextBuffer` outside of a `TTest` derived class:

```
void TBar::DoStuff(short foo) // This is NOT a derived class of TTest
{
    TTieredTextBuffer cout;
    cout << "Foo = " << foo;
}
```

If you are writing to a `TTieredTextBuffer` from multiple threads in the same task concurrently, you must use a `TChunkyTextBuffer`.

COMBINING TESTS

You have three ways to group the tests you have developed:

- ✦ Write a script that performs multiple tests.

A script allows you to create groups of tests that run sequentially.

- ✦ Combine multiple operations into a single test class.

This approach allows you to use a single instance that allows you to test several functions. Using a single TTest object allows the decision functions to share code or member functions.

- ✦ Group multiple TTest objects into a single test.

The approach allows you to run tests in a specific sequence or to run the tests in a random order.

Using a script to run multiple tests

You can combine tests that need to run together as a group into suites. Each test suite has a script associated with it that runs all the tests. This script is currently an ASCII text file that usually calls RunTest many times.

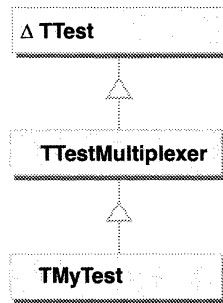
Combining operations in a single test class

Your TTest class can become very large and complicated if you try to test many member functions in the Test member. A simpler way to test multiple member functions inside a test class is to create a derived class of TTestMultiplexer.

TTestMultiplexer contains a number of decision functions. These decision functions are methods analogous to the TTest::Test member function. They are different in that they return a Boolean result rather than calling TTest::SetSuccess. A TRUE return value indicates success. Each decision function has an associated text key. This key is used to select the decision function at runtime.

You cannot manipulate the member functions inside a `TTestMultiplexer` as if they were subtests. However, using the `TTestMultiplexer` derived class allows the decision functions to share code or members.

FIGURE 7
DERIVE YOUR TEST
CLASS FROM
`TTestMultiplexer`
IF YOU NEED TO
PERFORM MANY
OPERATIONS IN A
SINGLE TEST CLASS



To make your tests easier to understand you should name the decision functions the same as the member functions. However, there is not always a one-to-one correspondence between decision functions and member functions.

Each decision function decides whether the associated member function in the target class works correctly. If some of the member functions in the target class are overloaded, you must differentiate the associated decision functions in the test class by giving them slightly different names.

You can use the `TTestMultiplexer` protocol to run all or some of the test class decision functions.

This means you can write fewer `TTest` derived classes if you consolidate several tests inside a single `TTestMultiplexer` derived class by placing the behavior of each test inside a member function, called a decision function. The decision functions can be executed by name using a text input to select the decision function to be executed in `Test`.

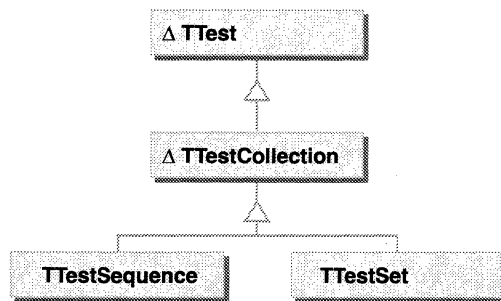
Combining multiple TTest objects into a single test

To make it easier to manage your instances of TTest, use the TTestCollection derived classes, TTestSet and TTestSequence, to group related TTest classes into a single test. The resulting test passes only if all of its subtests pass. This approach allows you to use your TTest classes individually.

In addition to providing the collection behavior, TTestCollection derived classes allow you to:

- Define the order subtests execute
- Shuffle the subtests into new random order
- Propagate inputs from the group to the subtests

FIGURE 8
DERIVE YOUR
TEST FROM
EITHER
TTESTSEQUENCE
OR TTESTSET TO
CREATE A GROUP
OF TTEST
INSTANCES



Every test in a group's collection is owned by the group. That means when a group is destroyed every test in the group's collection is also destroyed.

Creating tests with dependencies on other tests

You can create tests that have different behavior depending on the outcome of other tests. For example, TSecondTest only works if the system is in a state that is only achieved if TFirstTest is run and passes.

To allow this, place TFirstTest, then TSecondTest inside a TTestSequence. When the TTestSequence executes, the subtests run in order. If a subtest fails, the remaining subtests do not run. This behavior can be switched on and off.

To ensure that your test has everything it needs to run, check the prerequisites by overriding Setup, and if you can't run a test, then throw an exception in Setup.

IDENTIFYING WHAT A TEST DOES

You can retrieve various types of meta-information, including the purpose of a test and the name of the class being tested, using the `CopyInfo` member function.

Associated with each test class are meta-information key-value pairs in a dictionary. This supports categorization of different kinds of tests and analysis of large numbers of test results.

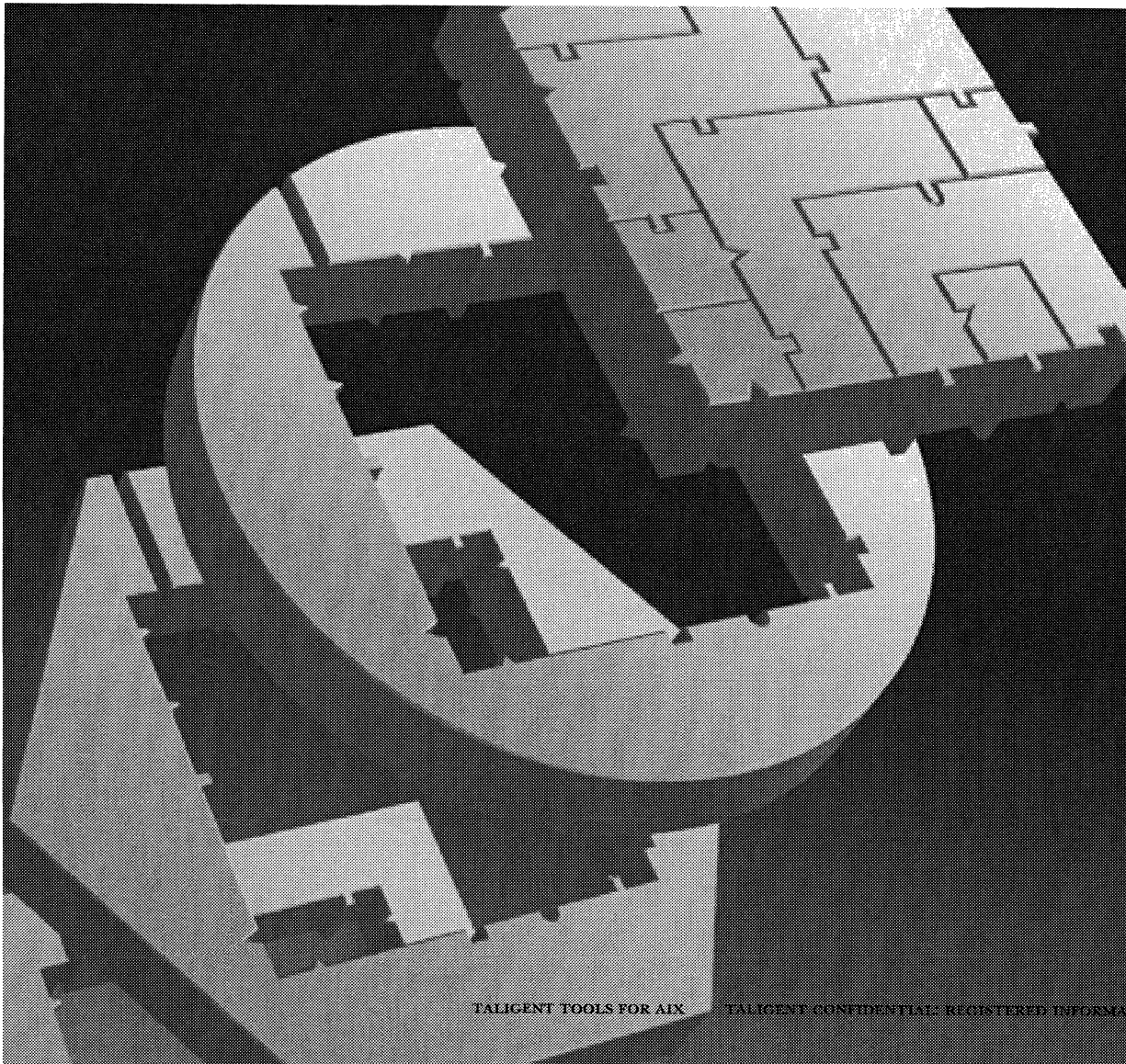
```
void TTestTestMultiplexerTest::CopyInfo(TDictionary& infoDict) const
{
    infoDict.AddKeyValuePair(new TStandardText(kDescriptionKey), new TStandardText(
        "A test for the TTestMultiplexer::Test member."));

    infoDict.AddKeyValuePair(new TStandardText(kInputSyntaxKey), new TStandardText(
        "TTestTestMultiplexerTest [-p <keyword> | -i <keyword...>]"));

    infoDict.AddKeyValuePair(new TStandardText(kTargetClassKey), new TStandardText(
        "TMultiplexerTest"));

    infoDict.AddKeyValuePair(new TStandardText(kTargetSharedLibraryKey), new
TStandardText(
        "TestLib"));
}
```

You can print a test in a textual form in order to browse a collection of tests before or after the tests are run.



CHAPTER 9

RUNTEST

One of the primary benefits of the Test environment is a uniform protocol for running tests. The ability to derive all tests from the abstract base class TTest provides this capability. Test framework users need not write applications to run their test—the program RunTest runs all tests derived from TTest.

RunTest provides the ability to instantiate a derived class of TTest in a newly created task, or in another task.

When you start RunTest, you must give at least one of the three options -test, -log, or -start. In addition to these options, RunTest also understands the options of TTest::CreateTest. CreateTest options specify what test to run. RunTest options specify how the test is to be run.

You can also create a script that launches RunTest multiple times to group your tests into a suite of tests.

PERFORMING A TEST

To perform a test:

- 1 Make sure that your tests are compiled and linked into a shared library.
- 2 Launch the RunTest application that runs a test, giving the name of the test class and the name of the shared library containing that class.

For example, You can launch RunTest from the console in order to execute a test named TMyTest in a shared library named TMySharedLibrary with the following command line. The `-e d` option increases the echoing level to “detail.”

```
> RunTest -t TMyTest TMySharedLibrary -e d
```

RunTest passes to your test as arguments any command line arguments following the `-o` option. For example, in order to pass the options called full and l to TMyTest:

```
> RunTest -t TMyTest -l TMySharedLibrary -e d -o full l
```

Testing an interface inherited from a base class

You can define the target of a test either in the test class or as a parameter to the RunTest program.

To test class TMyClass, write TMyClassTest, which contains an instance of TMyClass as its target. You can attach this instance in your TMyClassTest::Setup member function.

To allow polymorphic testing, use `-target` option of RunTest. For example, if you later make a derived class of TMyClass called TMyClassSubclass, then you can test it using TMyClassTest with no recompilation by typing:

```
> RunTest -test TMyClassTest MyClassTestLib -target TMyClassSubclass MyClassSubclassLib
```

PROVIDING INPUT FOR A TEST

Inputs to a test are textual (TText) so you can easily specify them in an application and so that a script interpreter can handle them.

A log contains a test's original inputs, which allows you to repeat tests with any inputs that cause a test to fail.

Pass input to a test as a collection of TText objects using SetInputs. These TText objects must be parsed, much like argv inputs to a main are parsed in an ANSI C program. The advantage of this method is that any application can pass inputs to a test, and inputs can be accurately logged after a test is run.

To see what text arguments a test requires, use TTest::CopyInfo to retrieve meta-information about a TTest derived class, including the input argument syntax.

If a test cannot function with a given set of inputs, it can fail with a diagnostic about unreasonable inputs.

Parsing text inputs to a test

To parse the inputs to the test, use the TTextArgumentDictionary class. TTextArgument takes as input an ordered collection of TText objects and parses the TText objects as arguments on a command line, forming key-value pairs.

TTextArgumentDictionary is a support class that is not specifically tied to TTest.

A leading hyphen character identifies keywords. Anything without a leading hyphen is a value argument. Keywords pick up the following argument if it isn't another keyword. For example, the next table shows how the command line input to a test is parsed.

Command line input:

```
-foo -bar squall8 fork1 spoon -ccc 84 85
```

Dictionary entries:

Key	Value
-foo	
-bar	squall8
1	fork1
2	spoon
-ccc	84
3	85

Note that:

- ※ All key and value objects are TText objects (actually, a concrete derived class of TText).
- ※ The value associated with -foo is an empty TText, not NIL. This allows clients to distinguish “There is no -foo keyword” from “There is a -foo argument with no associated value.”
- ※ The -bar argument picks up the following argument, squall8, as its value.
- ※ Arguments fork1, spoon, and 85 have no associated keyword argument and are assigned keys of 1, 2, and 3, in the order in which they appear in the input collection. These keys are TTexts not numeric values.

You can specify that certain keywords never take value arguments. Such keywords are called naked options.

You can also specify that certain keys can take more than one argument. These are called multiple-value options. Another example follows where -x is defined as a naked option and -values and -libs are defined as multiple-value options.

Command line input:

```
-g 40 -x 43 -values 1 4 33 -z 4 -libs Foo Bar
```

Dictionary entries:

Key	Value
-g	40
-x	
1	43
-values	1, 4, 33
-z	4
-libs	Foo, Bar

Note that:

- ※ Even though -x is followed by a value, because it is a naked option, it does not take the value. Instead, the following value is taken to be a key-less option.
- ※ The -values and -libs arguments are able to take more than one value, because they have been specified to be multiple-value options.

RUNTEST OPTIONS

RunTest Option	Description
-a[sync]	Run test asynchronously, do not wait for test completion. This option applies only if -s or -m is given as well. When running tests on a server, RunTest normally makes a synchronous call. If you specify -a, RunTest instead queues the test on the server and returns immediately.
-d[ebug]	RunTest breaks into the debugger just before calling the TTest::Run member function of the test.
-i[nteractive]	By default, tests are not interactive. They do nothing that requires human interaction (no dialog boxes, no breaks into the debugger). If you give the -i flag, the test is set to interactive mode, and is then able to do user interactions.
-lo[g] on off	RunTest turns global logging on or off for this machine. This affects all subsequent tests that are run with RunTest on this machine for this session, starting with the current test.
-m[achine] <i>name</i>	RunTest runs the test on the named machine. This is functionally similar to specifying -s directly on the remote machine.
-n <i>numberOfRuns</i>	The -n option specifies the number of times the test is to be run.
-o[ptions]	Ignores any arguments following -o and passes them to the TTest object by calling TTest::SetInputs.
-s[erver] [<i>name</i>]	Run the test within another test server. The server must be specified by the server name. If you do not give a name, RunTest uses the named server RemoteTestServer communicating through message streams.
-st[art]	RunTest starts the test server in its own task and remains active indefinitely (normally RunTest terminates immediately after test completion). You must specify the -s option with a server name. This server name uniquely identifies the test server.

CreateTest options interpreted by RunTest	Description
<code>-t[est] class library</code>	The TTest derived class to be instantiated and the shared library of the class. CreateTest instantiates an object of class <i>class</i> using the default constructor. CreateNewObject cannot instantiate the test by name if the derived class has either an inline default constructor or a compiler-supplied default constructor.
<code>-e[cho] h g n d D</code>	Set detail of diagnostic output: headline, general, normal, detail, Debug. This option specifies the echo level for printing the diagnostic output from the tests. TTest::OutputTextStream selectively echoes text to the console depending on the echo level.
<code>-ta[rget] class library</code>	Specify the class and shared library to be the target of the test. Instantiates an object of the class using the empty constructor. Use this option when the test you are running requires that the caller setup the target before the Run member function is called.

STOPPING A TEST

To stop a test from inside the test, raise an exception—any exception that you do not catch yourself. This is caught by the Test framework and terminates the test. This also causes your test to fail, because all tests that terminate by exceptions are defined to fail.

To stop a test from outside a test, you must terminate the task running the test.

EXAMINING TEST RESULTS

PrintTestReport gives you the ability to see the test results for tests that have been logged using the -log option in RunTest. PrintTestReport accepts the following options on the console command line:

-e[cho] h g n d D	Set detail of diagnostic output: headline, general, normal, detail, Debug. This option specifies the echo level for printing the diagnostic output from the tests.
-f[ail]	Show failing tests only. By default, prints all tests that have been logged.
-file <i>fileName</i>	Use the log file <i>fileName</i> .
-k[ey] <i>key</i> [<i>value</i>]	Specify a key and a value to retrieve a more specific subset of the tests. For example, you could search the log for all tests with key = kTargetSharedLibraryKey and value = HighLevelToolBox, to retrieve all tests run on high level Toolbox. Define the key-value pairs for a test in the CopyInfo member function.
-p[ass]	Show passing tests only. By default, prints all the tests that have been logged.
-s[ummary]	Print a summary for all the tests that were logged. The summary includes the total number of tests, the number of tests that passed, and the number of tests that failed.

Collecting timing information

Tests run for a finite amount of time. You can analyze the total elapsed time to run a test to conduct performance testing.

You can also identify the times when a test started and stopped to determine if tests are running concurrently and might affect each other.

HANDLING EXCEPTIONS

If a software exception occurs in the Test member function, the Test framework catches the exception and handles it.

To be notified when an exception occurs, override TText::HandleException, which is called whenever an exception is caught.

If a hardware exception or fault occurs in the Test member function, a monitoring task can terminate the task running the test.

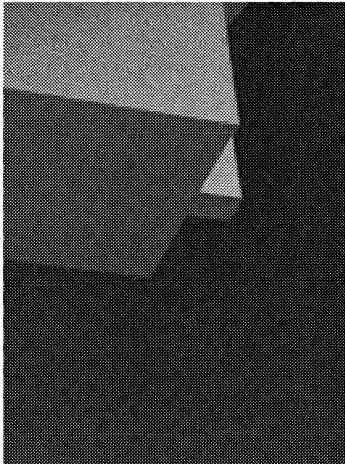
PART 3

SNIFF+ GUIDE

Chapter 10 Getting started	127
Chapter 11 Using SNIFF+	145
Chapter 12 Basic Elements	169
Chapter 13 SNIFF+ subsystems	185
Chapter 14 Customizing your environment	231
Chapter 15 Support for other functions	249

Appendix B
GNU Regular Expressions 257

Appendix C
ETRC file entries 261

**SNiFF+ GUIDE**

Copyright © 1994 Taligent, Inc. All rights reserved.

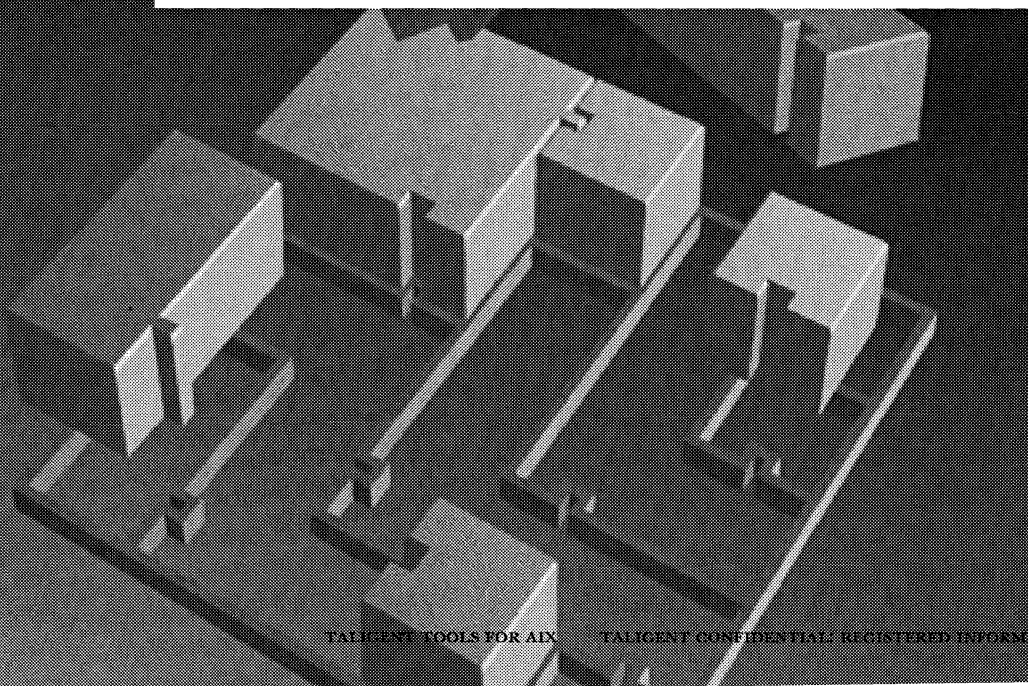
Copyright © 1994 takeFive Software, Inc. All rights reserved.

This manual is copyrighted. Under the copyright laws, this manual may not be copied, in whole or part, without prior written consent of Taligent or takeFive.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

This product may be protected by one or more U.S. and International Patents.

TRADEMARKS: Taligent and the Taligent Design Mark are registered trademarks of Taligent, Inc. SNiFF+ is a trademark of takeFive Software, Inc.



CHAPTER 10

GETTING STARTED

NOTE This documentation is work in progress.

- ✦ The tutorial uses ET++ instead of Taligent examples.
- ✦ Cross-references have not all been updated and the chapters have not been edited.
- ✦ References to the license file do not apply.

If the SNIFF+ instructions appear to contradict information in earlier parts of the *Taligent Tools for AIX*, follow the information in the earlier chapters.

INTRODUCTION

The SNIFF+™ open environment provides browsing, cross-referencing, design visualization, documentation, and editing support. It delegates compilation and debugging to any C++ compiler and debugger of choice.

ABOUT SNIFF+ DOCUMENTATION

The SNIFF+ documentation set consists of the following chapters:

- Getting Started
- Using SNIFF+
- Basic Elements
- SNIFF+ subsystems
- Customizing your environment

“Getting Started” and “Using SNIFF+” provide step-by-step introduction to SNIFF+. Using a real-world software system, they guide you through the various tools and show you how to use them efficiently.

Both newcomers to C++ or programming environments and experienced programmers should read the first two tutorial chapters to learn about the underlying principles and provides background information important using SNIFF+ tools.

The remaining reference chapters provide a complete and concise description of all SNIFF+ tools and menus.

Time needed to work through this manual

To only read the two tutorial chapters, you will need around 1 1/2 hours. To work through and follow all steps in front of a workstation, you will need roughly 3 hours.

Examples used throughout this manual

All examples are taken from the ET++ public domain class library, whose source is part of the SNIFF+ software distribution. ET++ is an object-oriented application framework developed by University of Zürich and the UBILAB of the Union Bank of Switzerland.

Experts know that ET++ is well designed and has a clean object-oriented programming style. The core library without examples has around 250 classes and has ca. 80K lines of code. SNIFF+ itself was built on top of an internal version of ET++.

Using ET++ as the basis for this manual allows you to get familiarized with SNIFF+ in a context that is very close to real-world software projects.

Feedback

Feedback is always very welcome. Send feedback to our e-mail address:

`sniff@takeFive.co.at`

Send feedback on class and member descriptions to Taligent.

Terminology

Before starting the actual description of SNIFF+ it is important to explain the meaning of some frequently used terms.

- ※ While SNIFF+ is a tool, it provides different kinds of tools itself. To simplify the text, we use the term tool for all tools SNIFF+ provides.
- ※ The exact distinction between the terms editor and browser has been blurred so much that they are sometimes used as synonyms. We use the term editor (e.g., the Project Editor) when we talk about a tool that is used for both viewing and changing data. We call a tool a browser (e.g., the Symbol Browser) when it is used for viewing only. All tool names are capitalized.

 **NOTE** The Documentation Browser can change data when editing is engaged (see “Documentation Browser” on page 224).

- ※ A programming environment deals with source code from which it extracts information, which it represents in several ways. This information consists mainly of data about declaration, definition, and use of named program elements such as classes, methods, variables, and functions. We call a named programming language construct a symbol. The repository where SNIFF+ stores the information about the symbols defined in a project is called symbol table.

Typographical conventions

- ※ Tool names, window names, and menu names start with capital letters.
Examples: Symbol Browser, File dialog, Icon menu.
- ※ Menu entries are enclosed in double quotes.
Example: Menu entry “Mark classes defining *method*”.
- ※ Placeholders for names of symbols, selections, or other strings are printed in *italic*.
Example: Menu entry “Mark classes defining *method*”.
- ※ Code examples and inputs that have to be typed in by the user are printed in monospace typeface.
Example: Type in: This text has to be typed in by the user.
- ※ Special keys are printed in Courier typeface with enclosing '<>'.
Examples: <Ctrl>, <Enter>, <Alt>.

BASIC SNIFF+ CONCEPTS

The main goal in developing SNIFF+ was to create an efficient and portable C++ programming environment that makes it possible to edit and browse large software systems textually and graphically. Much emphasis was placed on run-time and memory efficiency and on a comfortable user interface.

Main tools

A running version of SNIFF+ consists of two operating system processes, the information extractor and the programming environment itself. The information extractor can run locally or on any node on a network. Its task is to extract information about definitions and declarations from the source code.

The programming environment consists of a number of tools that are organized around a kernel consisting of the symbol table and the project manager. Both the symbol table and the project manager organize information in main storage for use by browsers and editors.

The symbol table manages the information about symbol definitions and declarations, and the project manager manages the information about open projects, such as the source files they consist of and various attributes.

Information extraction


The SNIFF+ information extractor is a fuzzy C++ parser. This means that it understands enough about C and C++ to extract the information of interest without having to understand C++ completely. This approach makes it possible to parse every file only once without including header files and expanding macros.

Not expanding macros is somewhat controversial because it could result in a loss of information if macros are used to change the syntax or semantics of C++. Experience with real projects shows that this is not a problem. Not expanding macros means that the symbolic information corresponds exactly to the locally visible source code. This is frequently an advantage, for example, when macros are used to put unique prefixes in front of all class names.

The SNIFF+ information extractor extracts information about declarations and definitions of C++ language elements and macros. It does not extract information about the usage of symbols. This information is extracted on the fly with the Retriever.

Updating information

If the source code of a project is edited, the information about the location of the affected symbols is updated immediately. On saving a file, its symbolic information is extracted anew and all browsing tools are updated. A user, therefore, always works with symbol-based tools presenting information that correctly mirrors the source code without ever having to bother about the effects of changes. This updating is done only if the SNIFF+ Editor is used


 **NOTE** If the source files are changed with external editors (e.g., vi), SNIFF+'s symbol table is updated next time the file is read.

Project concept

To start working with SNIFF+, a developer has to define a project. A project consists of a set of source files and, possibly, a set of subprojects that can be shared among projects. A subproject is a complete project on its own.

A typical project structure for a program building on a class library is to have a root project containing the project-specific (application-specific) source files and to load the library project as a subproject. Library projects are frequently trees of projects themselves.

Whenever a project is opened or a file or a subproject is loaded into the current project, its source code is analyzed and the information about the symbols defined therein is stored in the SNIFF+ symbol table.

 **NOTE** Software systems (like InterViews) that store implementation and header files in different directories can be handled best with SNIFF+ by creating separate projects for the implementation files and for the header files. The Implementation file project is then loaded into the header file project as a subproject.

Browsers and editors

Once a new project is defined with the Project Editor or an existing project is opened, it can be browsed and edited in different ways.

Symbol Browser

The Symbol Browser provides an overview of symbols defined in the source code; it displays the results of queries sent from other tools (e.g., “list all symbols matching a certain name”).

Class Browser

The Class Browser can be used to browse through the locally defined and inherited elements of a class.

Hierarchy Browser

The Hierarchy Browser displays the inheritance hierarchy and visualizes queries such as “mark all classes declaring method Add()”.

- Retriever** The Retriever can be used to obtain information about where a certain symbol is used in the source code (i.e., cross-reference information). The Retriever is a text-search-based tool. It makes it possible to extract all occurrences of strings matching the name of a symbol (or any regular expression) in a set of projects and to apply semantic filters to the matches.
- Editor** SNIFF+ has two possibilities for editing:
- ⌘ The integrated Editor is a mouse- and menu-driven Editor. It understands C/C++ syntax, provides browsing support, and automatically highlights structurally important information such as class names, method names, and comments. When a source file is modified, it is possible to trigger its compilation from the Editor and to mark the source lines where the compiler found syntax errors.
 - ⌘ The Emacs 19 editor has symbol highlighting. This manual uses the integrated Editor for all examples. Please refer to “Emacs integration” on page 250 for a description of how to integrate Emacs.
- Documentation Browser** The Documentation Browser lets you view and edit class and member descriptions.

-
- Shortcuts** The complete functionality of SNIFF+ is provided in the menus of the various tools. To speed up the work, especially for experienced users, SNIFF+ provides three different types of shortcuts to allow faster access to the commands found in menus.
- ⌘ Keyboard shortcuts are issued by holding down <Alt> of your keyboard and pressing the key that is shown at the right of a menu entry. Throughout this manual we work with the menus rather than keyboard shortcuts for command selection. Some frequently used shortcuts are:
 - ⌘ <Alt>C for copy
 - ⌘ <Alt>V for paste
 - ⌘ <Alt>B for browse class
 - ⌘ Mouse shortcuts are issued by double-clicking with the mouse on entries in lists or selectable items. Throughout this manual mouse shortcuts are used wherever possible. Some frequently used shortcuts are:
 - ⌘ Editing the source of a symbol by double-clicking on it in the Symbol Browser
 - ⌘ Jumping to the source location of a variable by double-clicking on it in the Class Browser

- ※ Deep clicks are issued by holding down the <Ctrl> key and pressing the left mouse button. Some frequently used deep clicks are:
 - ※ Switching from the declaration of a symbol to its implementation by <Ctrl>clicking on the symbol in the symbol list of the Editor
 - ※ Restricting the information shown in the list of a Symbol Browser by <Ctrl>clicking on the checkbox of a project in the project tree view
 - ※ Showing methods of only one class in the Class Browser by <Ctrl>clicking on the class in the inheritance graph view

Fast positioning in lists Pressing a key while the mouse pointer is over a list will position the list to the first entry whose name starts with that letter.

PREREQUISITES

The following sections describe how the environment must be for you to use this manual and run SNIFF+ successfully.

Installation

The SNIFF+ product package is installed automatically when the entire Taligent product is initially installed. The source directory for all SNIFF+ files is `$TALIGENTROOT/$TOOLS/SNIFF`. For more information regarding the installation, see the Taligent installation guide or ask your system administrator.

Checking the environment

SNIFF+ needs two environment variables. The environment variables should already have been set by your system administrator. The following instructions show you how to verify their values, and to correct them if needed. If the variables are not set correctly, you set them in your `.login` or `.cshrc` file.

NOTE The environment variables and license file are set as part of the installation procedure. Use the following settings for reference—you should not need to complete the procedures.

`$SNIFF_DIR`

In the shell type

```
ls $SNIFF_DIR
```

If you see a list of files containing `bin`, `examples`, `doc` and some others then the variable is set correctly.

If not, set the variable by typing in the shell

```
setenv SNIFF_DIR <sniff_directory> (for csh)
SNIFF_DIR=<sniff_directory>; export SNIFF_DIR (for sh or ksh)
```

where <sniff_directory> is the root of the directory tree of your SNIFF+ installation. You can get the location from the Taligent installation guide or the person who installed the Taligent product (normally the system administrator).

\$PATH


In the shell type

```
echo $PATH
```


If you can see <sniff_directory>/bin somewhere, then it is OK.

If not, set the path by typing in the shell

```
set path = ($SNIFF_DIR/bin $path) (for csh)
PATH=$SNIFF_DIR/bin:$PATH; export PATH (for sh or ksh)
```

 **NOTE** You will not be able to compile and debug the example applications in this manual. Therefore the executable search path does not contain compiler or debugger names.

\$LM_LICENSE_FILE

 **NOTE** \$LM_LICENSE_FILE is set automatically by the Taligent installation procedures. The following information does not apply to Taligent users.

The \$LM_LICENSE_FILE variable has to point to a valid license file.

The license file can also be specified with the -c command line option of sniff. The following setting shows a configuration where the license file is located in the SNIFF+ installation directory:

```
setenv LM_LICENSE_FILE <sniff_directory>/license.dat (for csh)
LM_LICENSE_FILE=<sniff_directory>/license.dat;
export LM_LICENSE_FILE (for sh or ksh)
```

Copying the example files to your local directory

You have to copy the example source files to your home directory because you will modify them during the following sessions.

In the shell type

```
sniff_copy_example
```

This shellscript creates the directory ~/filebrowser and copies the files into it.

STARTING THE SNIFF+ TOOL.

SNiFF+ command line The command line syntax of SNiFF+ is

```
sniff [-c <license_file>] [<project_file>]
```

where the optional <project_file> is the name of an existing SNiFF+ project file and <license_file> points the license file to be used. Please refer to the Installation Guide for more information on licensing issues.

Starting SNiFF+ from shell

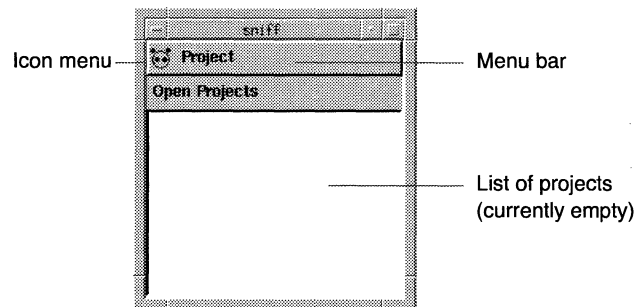
In the shell where you checked the environment variables type

```
sniff
```

or

```
sniff &
```

SNiFF+ should come up and you should see the empty Workspace Manager window:



After you complete the tutorial

You can open and browse the Taligent SNiFF+ projects.

Taligent.proj is the Taligent API interfaces project.

Stockbrowser.proj and ConcurrentGraphics.proj are Taligent documented samples

When you create your own projects (perhaps by copying one of the Documented Samples), add Taligent.proj as a subproject.

CREATING A NEW PROJECT

In this section you will create the filebrowser project, load the already existing source files into SNiFF+, and add the ET++ project as a subproject. The ET++ project is loaded as a subproject because the filebrowser project is based on ET++.

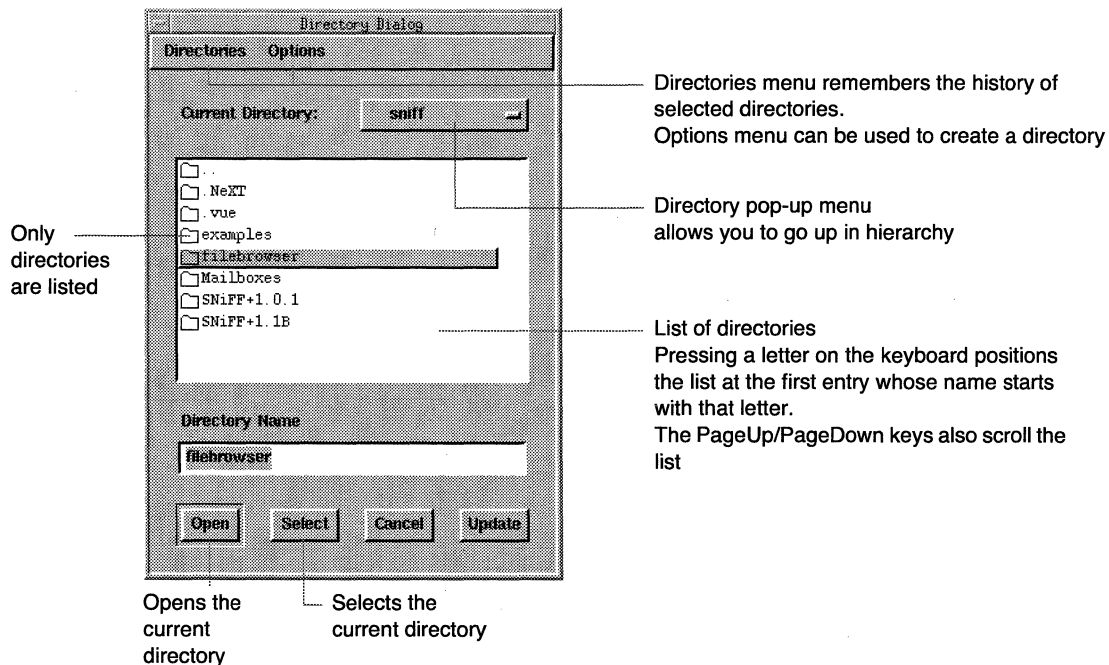
Creating the filebrowser project


- 1 Choose “New...” from the Project menu of the Workspace Manager.

A Directory Dialog is opened, prompting you for directory where the source files of your project are located.

- 2 Select the ~/filebrowser directory in the file list by clicking on it.

You can navigate either using the file list of the Directory dialog, by clicking at the directory pop-up menu, the Directories menu, or by manually typing in the directory name.



 **NOTE** Each Directory dialog and File dialog expands C-shell metacharacters such as '~' (for the home directory) or environment variables (for example, \$SNIFF_DIR).

3 Press the Select button (If you are in the directory and can see the source files, type '.' and press the Select button).

4 Press "Yes" to load all C/C++ files located in the directory.

A Project Editor is opened, all sources files are parsed, and the symbolic information is loaded into SNIFF+.


After the files are loaded you are asked whether to save the yet untitled project.

5 Press the Yes button.

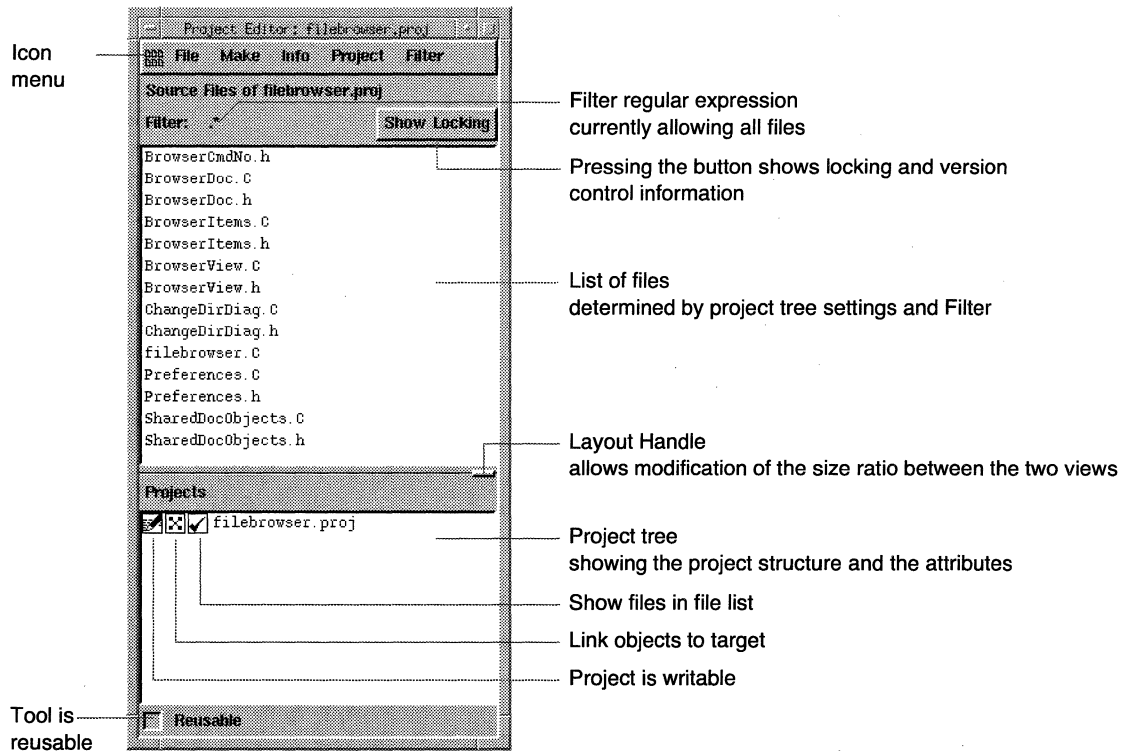
A File dialog is opened, prompting you for the name of the project file.

6 Position the mouse pointer on the text field of the File dialog and type in:
filebrowser.proj

7 Press the Save button or <Enter>.

 **NOTE** A project file stores only structural information and attributes of your project. No source code or symbolic information is stored there.

The Project Editor should look like this:

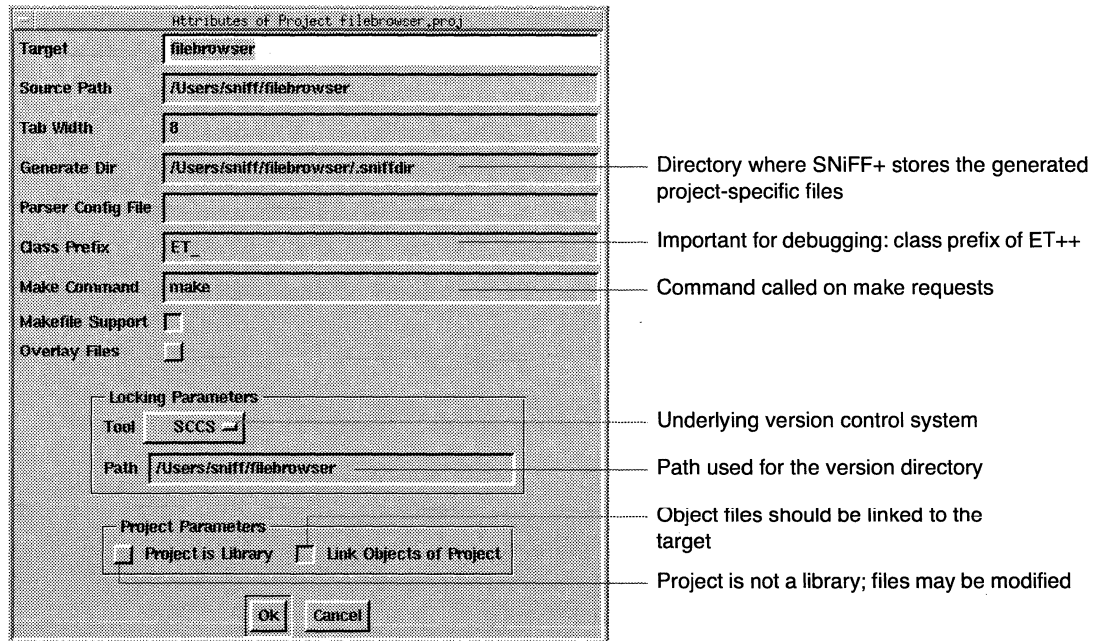


Setting the project attributes

Next you have to specify some attributes for this newly created project.

- 1** Select the newly created project in the project tree by clicking on it.
- 2** Choose "Attributes of Project filebrowser.proj" from the Project menu or double-click on the project in the Project tree.

A Project Attributes dialog is opened.



Defining the target name, tab length, class prefix and version control system

The target name is important because SNIFF+ uses it to drive the compiler and the debugger. The tab length has to be set to 8 because ET++ was developed with that setting and the SNIFF+ default value is 4 (see “Preferences” on page 231).

- 1 Type filebrowser in the target field.
- 2 Press the <Tab> key twice to set the keyboard focus to the tab width field and type in 8.
- 3 In the class prefix field type ET_.

NOTE In order to debug the example application, the class prefix field must be filled out correctly

- 4 Select “RCS” from the Tool pop-up menu of the Locking Parameters. Another possibility is “SCCS”, but Taligent does not use it.
- 5 Press <Enter> or click on the OK button.

Checking the source files into the version control system

You have chosen “RCS” as the underlying version control system. To work with the version control features of SNIFF+, you have to check in the files.

- 1 Press the “Show Locking” button.

The Project Editor changes its layout and shows the additional components for the version control system.

- 2 Press the “Select All” button.

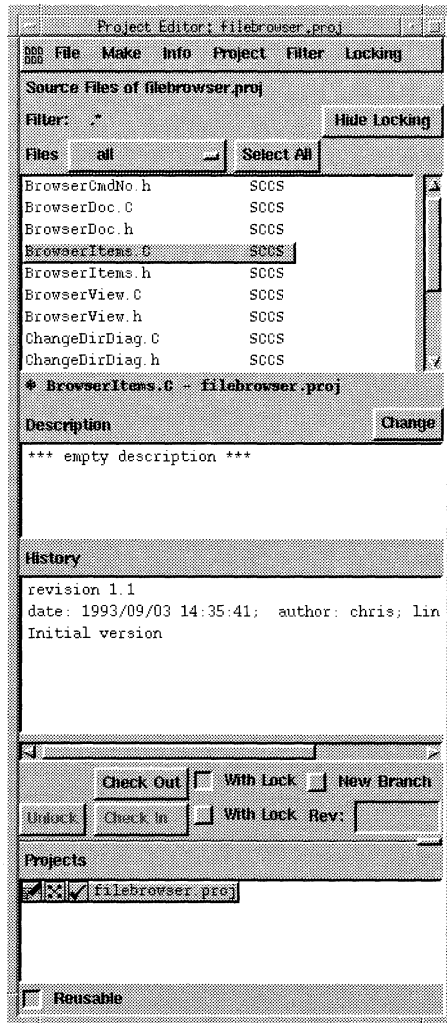
All entries in the list are selected.

- 3 Press the “Check In...” button.

A Log Message dialog is opened, prompting for a message to be stored with the initial version.

- 4 Type in Initial Version and press “OK”.

The Project Editor should look now like this:




- 5 Press the “Hide Locking” button.

The Project Editor is switched back to the initial mode and hides the version control information.

Loading a subproject

The filebrowser is based on the object-oriented application framework ET++, which is in the public domain and is also part of the SNIFF+ distribution. ET++ was already loaded into SNIFF+, so there is an already existing project file.

 **NOTE** Project structures must be created bottom up; i.e., you can load a subproject only if it is a SNIFF+ project itself.

- 1 Choose “Load Subproject...” from the Project menu of the Project Editor (filebrowser.proj must still be highlighted).

Now you see the File dialog prompting you for the project file.

- 2 Select `$SNIFF_DIR/examples/projects/et.proj` and commit.

The symbolic information for ET++ is loaded and the project tree of the Project Editor shows the new project structure. Since ET++ itself has a subproject called CONTAINER.proj, this structure is also shown in the project tree.

Investigating the attributes of the ET++ subproject

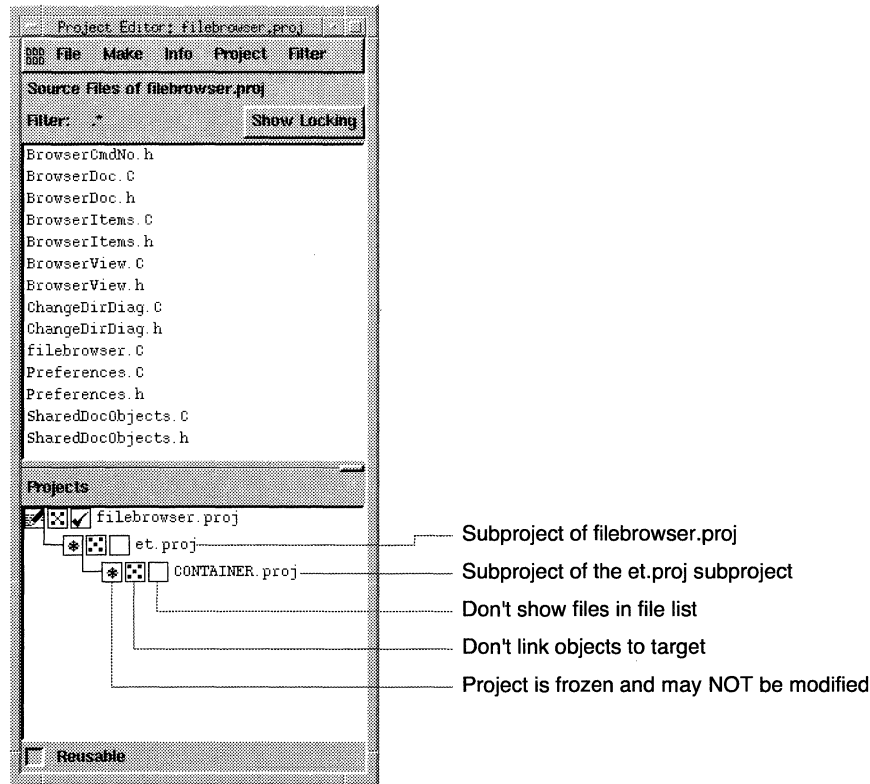
- 1 Select `et.proj` in the project tree.
- 2 Choose “Attributes of et.proj...” from the Project menu.

The Attributes Dialog for `et.proj` is opened.

You are not allowed to change the parameters because `et.proj` is a frozen project, meaning that no files may be modified. You can also see that the object files of this project should not be linked to the target.

- 3 Close the Attributes dialog by clicking the “Cancel” button.

Examining the results Now your Project Editor should look like this:



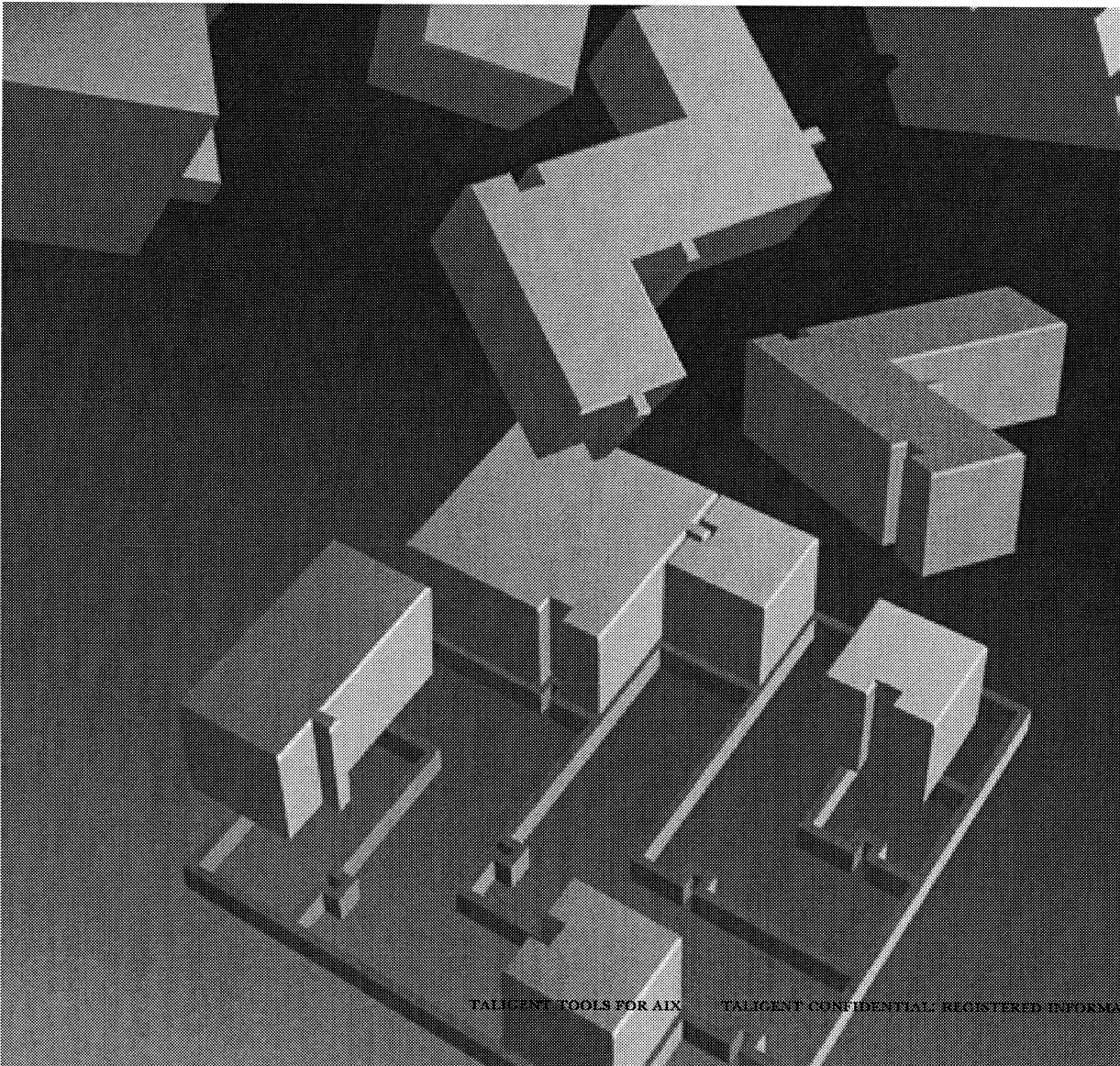
If not, close the newly created project in the Workspace Manager and restart from scratch.

Saving the new project and closing the Project Editor

The project creation is finished now and all that is left to do is to save the project specifications to a project file. From then on the Project Editor is only needed when the structure changes or attributes have to be edited.

- 1 Choose "Save Project filebrowser.proj" from the File menu of the Project Editor.
- 2 Choose "Close Tool" from the Icon menu.

The only open window now should be the Workspace Manager.



CHAPTER 11

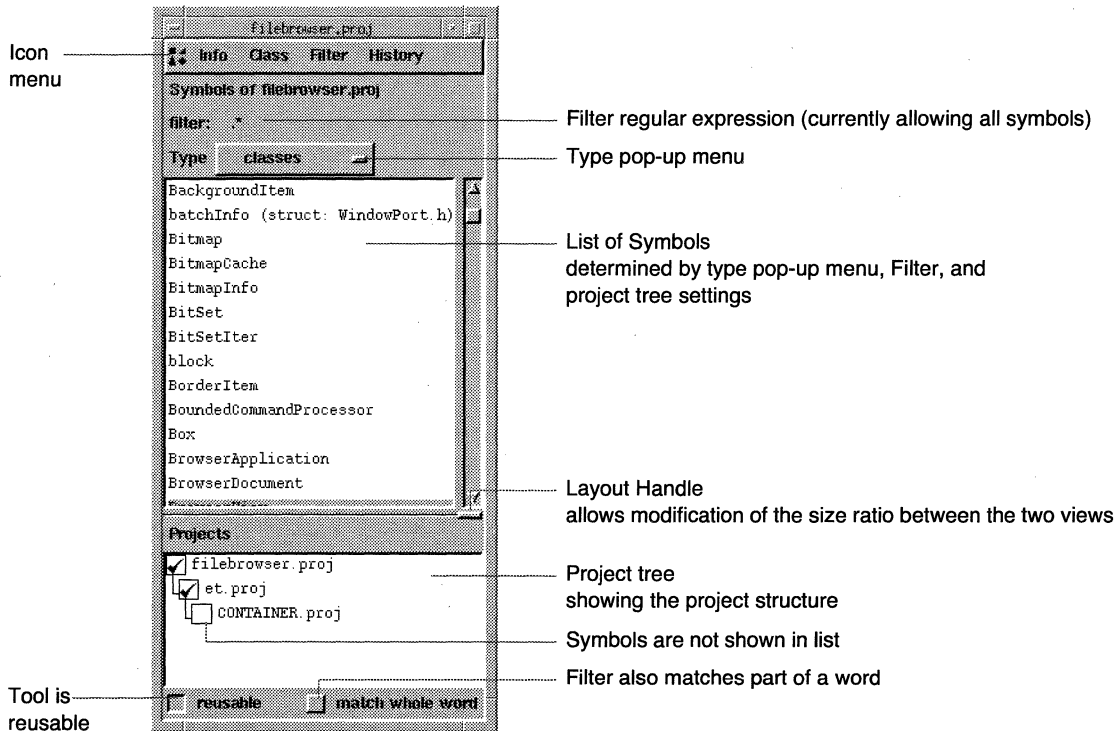
USING SNIFF+

BROWSING SYMBOLS

The handling of the Symbol Browser, the Class Browser and Retriever is very similar. Therefore the common parts of the browsers are described once while working with the Symbol Browser.

Opening a Symbol Browser

- 1** Select the `filebrowser.proj` in the Workspace Manager by clicking on it.
- 2** Choose “Symbol Browser” from the Icon menu.
A Symbol Browser is opened listing all classes.
- 3** Click on the `et.proj` check box in the project tree at the bottom of the window.
The classes of `et.proj` are now also listed.



Generally speaking, the Symbol Browser shows a list of symbols which is determined by the type selector, the project tree, and a regular expression matching the names of the symbols.

Relation to the source code

Every symbol of the list is defined in the source code. You can jump to the position in the source code defining the symbol by double-clicking on it.

- 1 Double-click on the symbol named `ActionButton`.

An Editor is now opened, the source files are loaded, and the cursor is positioned to the location defining/declaring the symbol.

NOTE Pressing a key in a list will position the list to the first entry whose name starts with that letter.

You will have a closer look at the Editor later.

- 2 Select "Close Tool" from the Icon menu to close the Editor.

Project tree

The project tree shows the hierarchical structure of the project the browser belongs to. The check boxes in front of the project names determine if the corresponding symbols are shown or hidden in the symbol list.

Restricting information shown in the symbol list

Symbols can be shown/hidden by directly manipulating the check boxes or issuing a menu command after selecting a project entry.

- 1** Switch the check box of the `et.proj` on and off and watch the results.
- 2** Choose “Select from all projects” from the Filter menu.
Now you see the classes of all projects, including the subprojects of `et.proj`.
- 3** Click on the `filebrowser.proj` entry (not on the check box of the entry). The entry should be highlighted.
- 4** Choose “Select From `filebrowser.proj` Only” from the Filter menu.
Only the classes of `filebrowser.proj` are displayed; the classes of all other projects are hidden. A deep click (`<Ctrl>`click) on the entry (not on the check box of the entry) gives the same result as the “Select from `filebrowser.proj` only” command.

Constraining the list with filters

SNIFF+ allows you to restrict the list of the browsers to entries matching a regular expression. This feature is very helpful when many entries are in the list and you want to focus only on a subset of them. The regular expression is also called a filter and conforms to the powerful GNU regular expression syntax (see “GNU Regular Expressions” on page 257).

Setting a filter

Let us set a filter to view only classes starting with the letter B.

- 1** Also view the classes of the `et.proj` by toggling the `et.proj` check box to on.
- 2** Choose “Set filter...” from the Filter menu.
A dialog pops up prompting you for the filter.
- 3** Type in `^B` and press `<Enter>` (the `^` in front of the B is correct and means beginning of line).
You see that only classes beginning with B are listed.

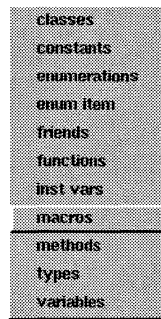
Because regular expressions are a powerful tool to limit the amount of information, they are also used in the Class Browser, in the Retriever, and in the Find/Change dialog of the Editor.

Resetting the filter

- Choose “Reset filter” from the Filter menu.
All classes are listed now.

Type pop-up menu

The Symbol Browser can show symbols of different types and macros. Besides classes, you can look at functions, friends, variables, types, etc. Methods can also be shown. (The list of methods can get very long, because it is a flat view of all methods of all classes if not further constrained).



- 1 In the project tree check the box of `et.proj` to show its symbols.
- 2 Try to show functions, macros, types, etc., by selecting different types from the type pop-up menu.
- 3 Switch back to classes.

TOP-DOWN BROWSING

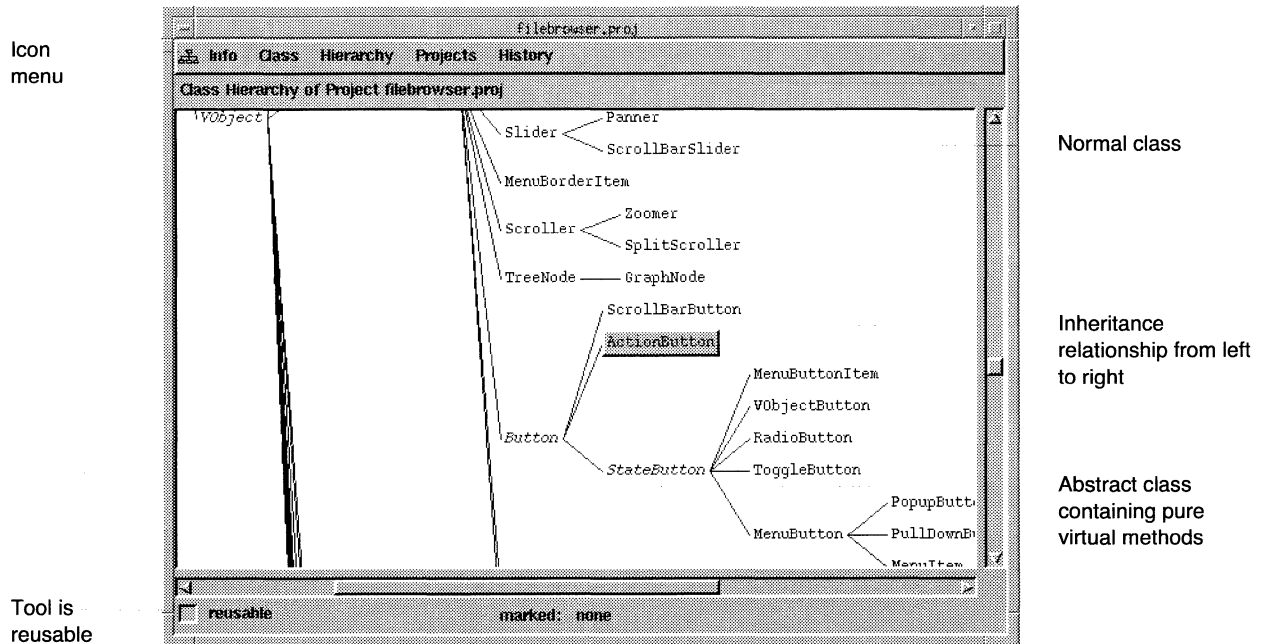
Top-down browsing is when you have a symbol, e.g., a class, and you want to learn more about its details and where and in what context it is used. Figuratively speaking, you are coming from a more distant view of your system (you just know that there is a class with that name) and are browsing down to the source code (bottom) and greater detail.

You will study this type of browsing now with the class `ActionButton`. During the following step-by-step tour you will start with the already familiar Symbol Browser and make acquaintance with the Hierarchy Browser, the Class Browser, and the Editor.

Viewing `ActionButton` in the class hierarchy

- 1 View all classes in the Symbol Browser (including classes from `et.proj`). You can do this by switching on the check box of `et.proj` in the project tree.
- 2 Select class `ActionButton` in the Symbol Browser.
- 3 Choose “Show Class `ActionButton` in Hierarchy” from the Class menu.

A Hierarchy Browser is opened showing the complete class graph and focusing on the class `ActionButton`.



- 4 Try to get an overview of the class hierarchy and the inheritance path of our class by scrolling around.

NOTE Although we use only single inheritance in our examples, the SNIFF+ tools also support multiple inheritance.

Loading all classes into the Hierarchy Browser allows you to get a good overview of the complete class hierarchy. However, you will find it very hard to follow an inheritance path up to the root without lots of scrolling. Therefore you will restrict the view to ActionButton.

Too much information—restrict information to ActionButton

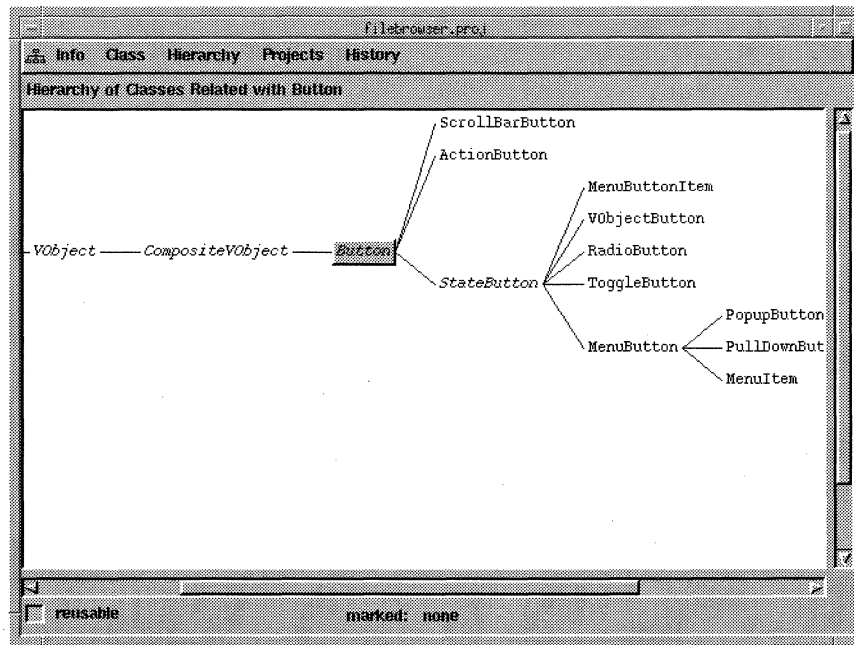
- 1 Select the class ActionButton in the Hierarchy Browser (if it is not already selected).
- 2 Choose “Show Class ActionButton in Restricted Hierarchy” from the Class menu.

Now the view is restricted to show only the superclasses and subclasses of ActionButton. All other classes are hidden. Since ActionButton is a leaf class, no other classes inherit from it. The information you get is now too limited. You get the best results for our purposes by restricting the view to the abstract base class of ActionButton, namely Button.

NOTE All tools print abstract classes in italic.

**Too little information—
restrict information to
Button**

- 1 Select the class Button in the Hierarchy Browser.
 - 2 Choose “Show Class Button in Restricted Hierarchy” from the Class menu.
- Now the view is restricted to show only the superclasses and subclasses of Button. All other classes are hidden. This gives you a better picture of the inheritance that leads to ActionButton and related classes.



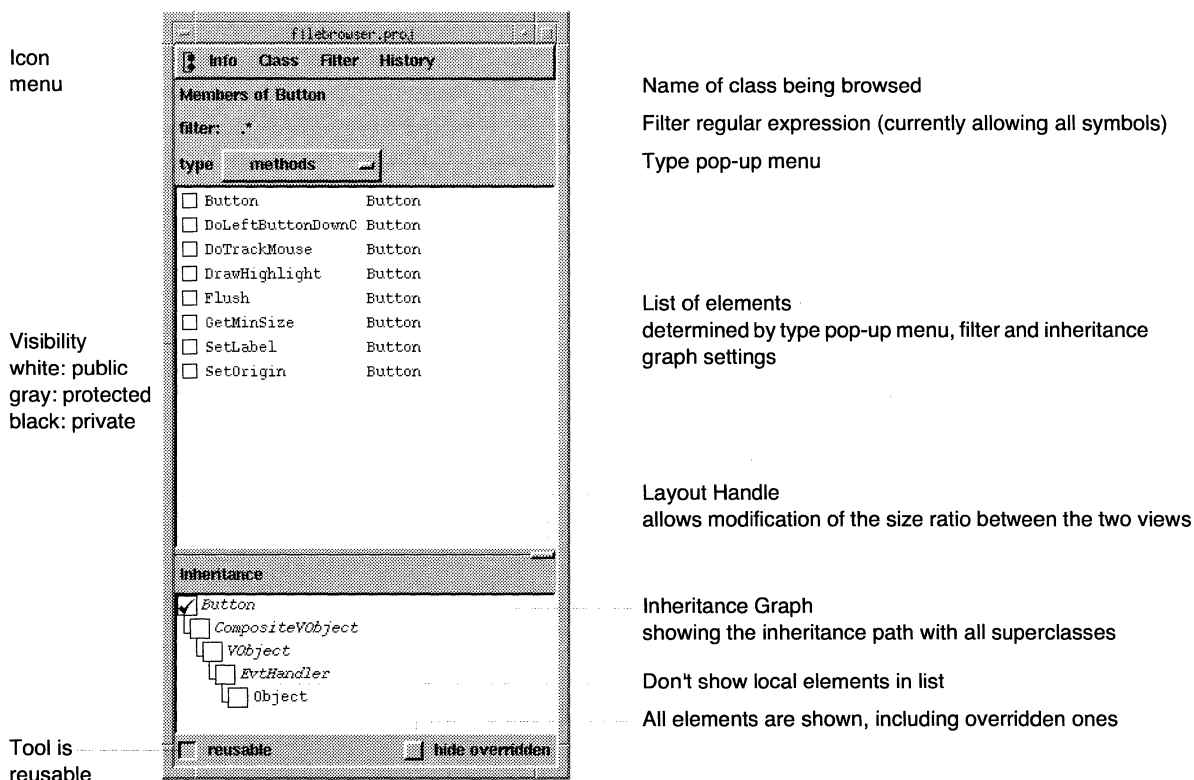
This is the context of our class in terms of inheritance. You will return to the class hierarchy later. Now you concentrate on the internals of ActionButton.

**Browsing the elements
of ActionButton—
the Class Browser**

With the Class Browser you can browse the internal structure of a class (in this manual local symbols of a class are called elements). The structure of the Class Browser is very similar to the Symbol Browser.

- 1 Select the class Button in the Hierarchy Browser (if it is not already selected).
- 2 Choose “Browse Class Button” from the Class menu.

A Class Browser is opened and the information about class Button is loaded.



Structure of the Class Browser

The Class Browser lists the elements of the current class identified by the element name and the name of the class defining the element. The small squares in front of the name show the visibility of the element

- ※ White is public.
- ※ Grey is protected.
- ※ Black is private.

Like the Symbol Browser, the Class Browser also has a type pop-up menu. Here you can choose among methods, instance variables, friends, types, or local enumerations of the current class. The list can be filtered with a regular expression.

Where the Symbol Browser has a project tree, the Class Browser has an inheritance graph reflecting the inheritance path. Each class can be toggled on or off individually.

- Check the box of class CompositeVObject in the inheritance graph.

You will recognize that not only the methods of class Button are displayed but also of class CompositeVObject.

What overrides what?

- Choose “Select From All Classes” from the Filter menu.

Now you have a completely flat view of the class `Button` including all overridden methods. Each entry in the list shows the method name and the class defining the method. So you see what overrides what. Method `Add`, for example, is introduced in `VObject` and overridden in `CompositeVObject`.

Hiding overridden methods

A completely flat view of the class is not always useful. Sometimes you want to see just the interface of the current class, hiding all the methods that are overridden.

- Press the button labeled “hide overridden” at the bottom of the Class Browser.

Now only the client interface of the loaded class is visible.

Studying protocols

Very often when browsing software systems you would like to know what overrides a certain method. SNIFF+ supports that type of protocol browsing by combining the Class Browser and the Hierarchy Browser.

- 1 Load the class `Button` into the Class Browser (if it is not already there).

You can do this by loading it from either the Hierarchy Browser or the Symbol Browser.

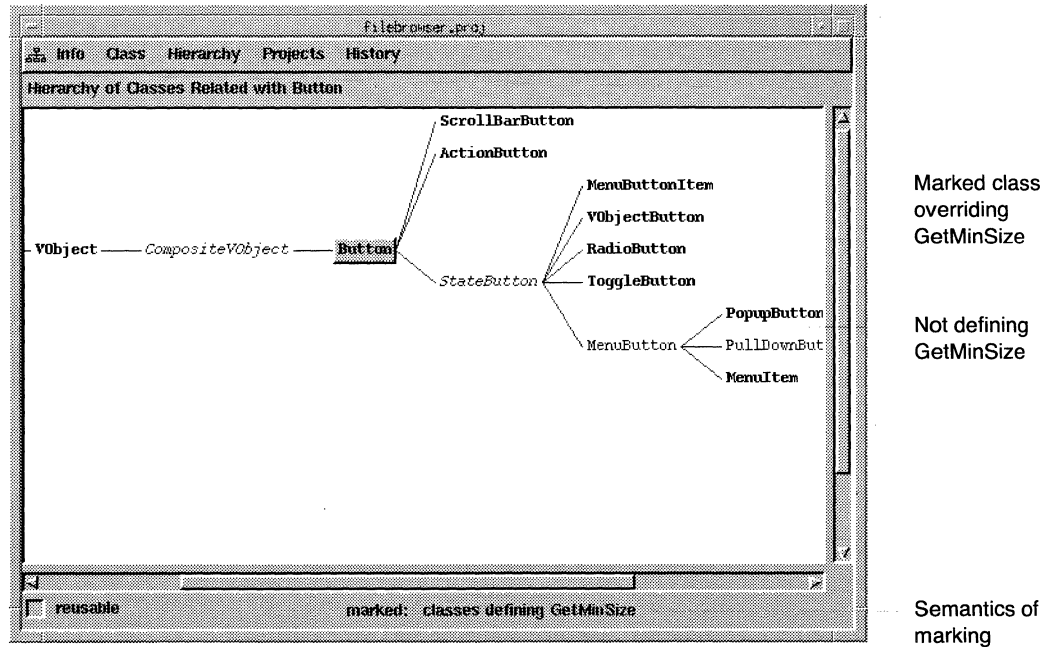
- 2 Select method `Button::GetMinSize`.

You can do this either by scrolling or by pressing the key 'G' (which positions the list to the first method starting with G).

- 3 Choose “Mark Related Classes Defining `GetMinSize`” from the Class menu.

The Hierarchy Browser is opened and all classes related to `Button` are loaded.

All classes displayed in boldface override the method `GetMinSize`. The Hierarchy Browser informs you about the marking in the status line.



Viewing the source code

By selecting boldfaced classes in the Hierarchy Browser, you can view the source code of the overridden methods.

- 1 Select ActionButton.
- 2 Choose “Edit Method GetMinSize” from the Hierarchy menu.
An Editor is opened, class ActionButton is loaded, and the cursor is positioned at the method implementation.
- 3 Try to look at the implementation of GetMinSize of some other classes in the Hierarchy Browser.

BOTTOM-UP BROWSING

Bottom-up browsing is when you start from the source code and you look at a symbol, e.g., a variable, and you would like to know more about its declaration and definition.

Figuratively speaking, you are coming from a special-usage context (source code, therefore bottom) and are browsing up to its declaration (higher view).

You will study this type of browsing, continuing where you stopped in the last section, namely with the implementation source code of `ActionButton::GetMinSize`.

During the following step-by-step tour you will start with the already familiar Editor and make acquaintance with the Retriever.

Studying the method `GetMinSize`

- 1 Load the source of `ActionButton::GetMinSize` into the Editor (if it is not already loaded). You can do that from the Class Browser, the Hierarchy Browser or the Symbol Browser.

- 2 Study the method.

You see that the variable `gLook` is used in the context of a method call.

The screenshot shows the Taligent Tools for AIX IDE with the following components and annotations:

- Icon menu:** Located at the top left, it reflects the editing state. Annotations include:
 - read-only
 - not modified
 - modified
 - currently: r/o
- Symbol names:** Printed in bold in the source code editor.
- Tool is reusable:** A checkbox at the bottom left of the editor window.
- Class pop-up menu:** Located on the right side of the class browser, it shows all classes or only one class (current setting). The current class is `ActionButton`.
- Symbol list:** A list of symbols defined by the class pop-up menu, including `ActionButton(mi)`, `Control(mi)`, `DrawInner(mi)`, and `GetMinSize(mi)`. Clicking on a symbol positions the cursor.
- Layout Handle:** A vertical bar between the editor and the class browser, allowing modification of the size ratio between the two views.

The source code in the editor is as follows:

```

Metric ActionButton::GetMinSize()
{
    if (TestFlag(eActionDefaultButton))
        return gLook->DefaultButtonLayout()->GetMinSize(this);
    return gLook->ActionButtonLayout()->GetMinSize(this);
}

void ActionButton::DrawInner(Rectangle, bool highlight)
{
    int code= 0;
    if (Enabled())
        SETBIT(code, 2);
    if (highlight)
        SETBIT(code, 3);
    if (TestFlag(eActionDefaultButton))
        gLook->DefaultButtonLayout()->Adorn(this, contentRect,
    else
        gLook->ActionButtonLayout()->Adorn(this, contentRect, c
  
```

What is gLook?

- 1 Double-click on the name gLook in the Editor.
- 2 Choose “Find Symbols Matching gLook” from the Info menu.
SNIFF+ opens a Symbol Browser and tries to find a symbol of any type matching gLook.

The Symbol Browser finds one symbol of type variable matching gLook. If there were any matches for other types, too, you could see this by clicking on the type pop-up menu. If no other entries are enabled, there are no other matches, which is the case for gLook.

Browsing the declaration of gLook

- Double-click on gLook in the Symbol Browser.
The source code declaring gLook is loaded into an Editor. gLook is a global variable and refers to an object of class Look. Now you should learn more about Look.

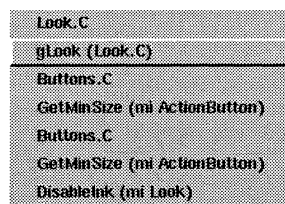
**Continued browsing—
what is Look?**

- 1 Select Look in the Editor by double-clicking on its name.
- 2 Choose “Browse Class Look” from the Class menu (the entry is enabled because SNIFF+ knows that Look is a class).
The class Look is loaded into a Class Browser.
- 3 Load the source code of some methods of Look into the Editor by double-clicking on entries in the Class Browser.
- 4 Close the Class Browser.

Going back in history

After browsing a lot, you seem to be lost somewhere in the source code. Didn't you start originally from the usage of variable gLook somewhere in `ActionButton`? Now you need the history feature of SNIFF+.

- 1 Click on the History menu of the Editor.
What you see in the pull-down menu are all the locations in the source code of our system you have visited during the browsing session.
- 2 Choose “gLook (Look.C)” from the History menu.
The Editor jumps back to the declaration of gLook.




An alternative to the history mechanism—tool locking

There is an alternative to jumping back and forth in just one Editor. By default every tool is reusable; that means whenever SNIFF+ needs a tool, it searches for an open tool of that type and uses it for the request. It opens a new tool only if there is no tool available. This feature prevents screen cluttering and too many open windows.

You can lock any tool of SNIFF+ against automatic (re)usage by releasing the “reusable” button in the status line. This feature is useful when writing code and simultaneously browsing two or more source files. Any browsing request will then open a new Editor.

- 1 Release the “reusable” button of the status line of the open Editor.
- 2 Double-click on any symbol in the Symbol Browser (currently only gLook is loaded, but you can load all classes into the Symbol Browser by choosing “class .*” from the History menu).

A new Editor is opened, leaving your locked Editor untouched.

 **NOTE** You can have as many instances of a tool as you like. After you lock a tool, SNIFF+ will not reuse that tool, but will open a new tool on a browsing request. It is good “SNIFF+ing style” to work with as few tools as possible

Where glook is used—the Retriever

In the previous session you started from `ActionButton::GetMinSize` and browsed the variable `gLook`.

Now you want to know where else in our software system this variable is used. The Retriever is a tool that allows you to find any matches in the whole project.

- 1 Load the declaration of `gLook` into an Editor (if not already there).

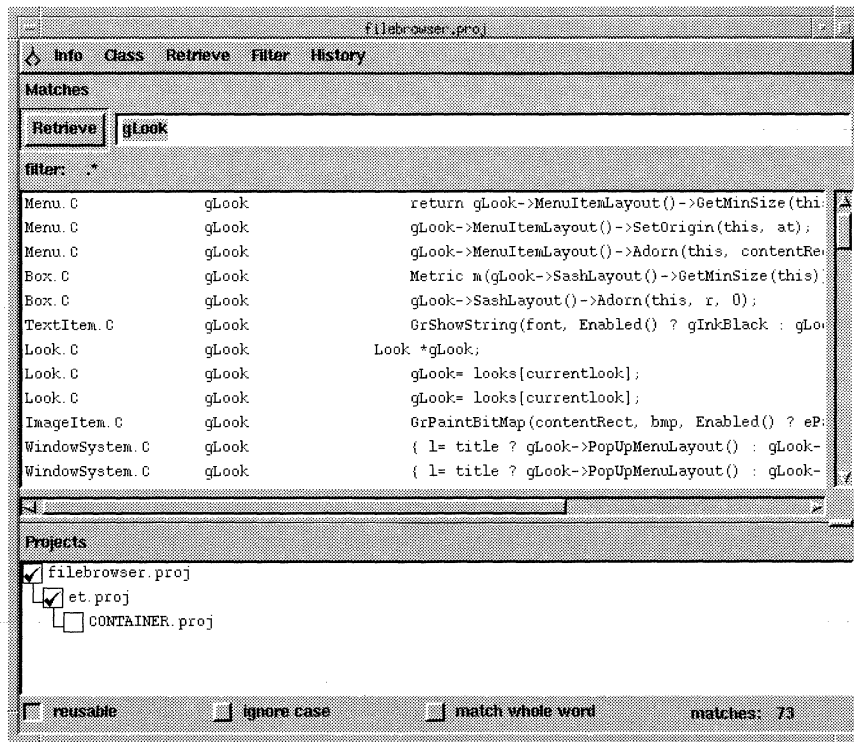
You can do this by double-clicking on `gLook` in the variable list of the Symbol Browser.

Retrieving gLook from all projects

- 1 Double-click on `gLook` in the Editor.
- 2 Choose “Retrieve `gLook` From All Projects” from the Info menu.

After a few seconds a Retriever is opened and the usages of `gLook` in our project, including all subprojects, are listed. This is too much information for us. Let's restrict the list to the places where `gLook` is assigned a value

Icon
menu



Search
string

List of matches
filename,
match,
source line

Layout
Handle
allows
modification
of the size
ratio between
the two views

Project tree

Also
search this
project
Tool is
reusable

Case sensitive Also match part of a word Number of matches

3 Choose “assignment” from the Filter menu of the Retriever.

What you see now are the two locations in our 60KLOC project where gLook is assigned a value.

The Retriever uses a two-stage filtering process:

- ⌘ At first all lines matching the search string are extracted.
- ⌘ Then the list is once more restricted by the regular expression (in this case a regular expression representing the syntax of an assignment).

NOTE The Retriever starts a full text search (like a super-grep in UNIX) over the project files and adds flexible semantic filtering as a second stage.

Of course, you can also load the code of the matches into an Editor.

4 Double-click on a match.

The source code is loaded into an Editor.


**Retrieving session 2—
getting information
about menu handling**

The Retriever is a very powerful tool for formulating fuzzy queries. Let's try to get all positions in our project that have something to do with menu handling.

1 Type Menu in the text field of the Retriever and press the Ignore case button in the status line.

2 Press the Retrieve button or <Enter>.

The Retriever lists hundreds of places (you can see the exact number in the status line). That's too many.

 **NOTE** After the first retrieve, the source code is cached and all further queries are much faster. You can switch caching off in the Preferences Dialog.

Let's apply the assignment filter.

3 Choose "Assignment" from the Filter menu.


Now you have the locations in your project where a variable called "menu" or similar is assigned a value.

**Where are menus
allocated on the heap?**

To get this information, you only have to apply another filter.


■ Choose "new" from the Filter menu.

Now you get about 60 locations in our project where a menu or something related is allocated on the heap

 **NOTE** The Retriever is a text retrieval tool with semantic filtering. It works best when the software system has consistent naming.

EDITING

SNIFF+ provides its own integrated Editor for editing source code. This section describes how to work with the integrated Editor. SNIFF+'s WYSIWYG Editor serves not only editing but also browsing purposes. It partially understands the C/C++ syntax and can format the text with different fonts and colors.

 **NOTE** Font and color settings for the formatted source code can be specified in the ETRC file. For more information see "Preferences" on page 231.

Loading a symbol into the editor

■ Load Class **BrowserView** into the Editor. You can do this by double-clicking the symbol in the Symbol Browser.

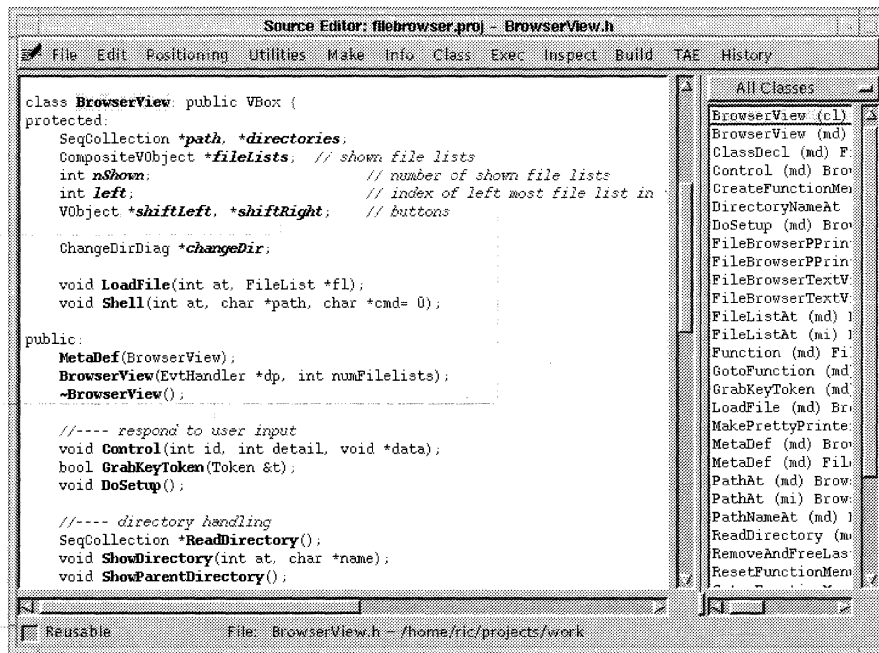
NOTE It is possible that you still have the **gLook** variable match in the Symbol Browser. If so, reset the filter and switch from the variable type to class type via the type pop-up menu

The class is loaded into an Editor and the cursor is positioned to the declaration of **BrowserView**.

Icon menu reflects editing state
 - read-only
 - not modified
 - modified
 currently: r/o
 Symbol names are displayed in special font faces

Comments are displayed with a special font face

Tool is reusable



Class pop-up menu either shows all classes or only one class (current setting)

Symbol list defined by class pop-up menu (clicking on a symbol positions the cursor)

Layout Handle allows modification of the size ratio between the two views

Because the **BrowserView** class belongs to the **filebrowser** project and the project is writable, you are allowed to modify this file. The status of the file is indicated by the tool icon at the upper left corner of the Editor.



File is read-only



File is writable



File is modified

Working with the Symbol list

The Symbol list shows the alphabetically ordered symbols (classes, methods and functions) of this file. It can be constrained by the pop-up menu just above the list. You can either select all classes or only one class. On a click on one symbol, the Editor immediately positions to the source code location defining the symbol. It also allows fast switching between declaration (normally in .h files) and implementation (normally in .C files).

- 1 Try positioning by clicking on various symbols in the Symbol list.
- 2 Choose “FileBrowserTextView” from the class pop-up menu above the Symbol list.
- 3 Switch back to the “BrowserView” context. You can do this either by selecting it from the class pop-up menu or by using the History menu.

Switching between declaration and implementation


With <Ctrl> mouse click on the symbol, you switch between declaration and implementation of the selected symbol.

- 1 <Ctrl> click on `BrowserView (md)` in the Symbol list.

The Editor now shows the implementation of `BrowserView::BrowserView`.

- 2 Try some other entries.

Checking out a file

 NOTE You can check out the file only if you have selected a version control system and checked in the files. See “Checking the source files into the version control system” on page 140.


The loaded files are read-only because we have checked in all project files before. To modify a file, you have to check out and lock the file.

- 1 Load the implementation of `BrowserView` into the Editor (file `BrowserView.C`).
- 2 Choose “Check Out” from the File menu.

The file is checked out and the editing state changes to writable. If you would open the Project Editor, you could see that the latest version of the file is locked by you.

Some useful editing helps

The Editor offers a lot of help that makes your life as a programmer easier. There are features like multilevel undo, matching brackets, nesting and unnesting, commenting and uncommenting, etc.

 **NOTE** SNIFF+ always keeps the locations of symbols in the source text up-to-date, even after inserting or deleting lines. If a modified file is saved, all tools will immediately update their views to reflect the newest set of symbols.

Matching brackets and quotes

- 1 Double-click to the right of the opening parenthesis of the last Add statement in `BrowserView::BrowserView`.

The Editor marks the text to the closing bracket.

```
Add(fileLists);
Add(new HBox(qPoint10, (VObjAlign) (eVObjHEqual|eVObjVBase),
  shiftLeft,
  new ActionButton (cIdChDir, "change directory . ").
  shiftRight,
  0
  ));
}
```

Nesting and unnesting, commenting and uncommenting

- 1 Select the last Add statement in `BrowserView::BrowserView` completely. You can do this by double-clicking left to the Add and dragging the mouse down to the last closing parenthesis while holding down the mouse button.


- 2 Choose “Comment” from the Edit Menu.

The complete statement is commented out.

```
// initialize left most file list
ShowDirectory(-1, ".");

Add(fileLists);
// Add(new HBox(qPoint10, (VObjAlign) (eVObjHEqual|eVObjVBase),
//   shiftLeft,
//   new ActionButton (cIdChDir, "change directory . ").
//   shiftRight,
//   0
//   ));
//
```

- 3 Undo the changes by choosing “Undo” from the Edit menu.

 **NOTE** SNIFF+ allows an arbitrary number of undo levels. The number of undo levels can be set in the ETRC file (see “Preferences” on page 231).

- 4 Don’t save the modifications you have made.

VIEWING AND EDITING CLASS AND MEMBER DESCRIPTIONS

In this section, you will begin working with a Taligent project. Before you begin, close the current project.

- 1 Use the Icon menu to go to the application window.
- 2 Close `filebrowser.proj`.

The Documentation Browser allows you to view and edit class and member function descriptions. Taligent source and documentation files are accessed from a prebuilt project called `Taligent.proj` which is located in

```
$TaligentSystemDocs/TaligentIncludesDocs/Public
```

The Taligent Application Environment class and member descriptions are stored in the `Docs` subdirectory of the `Public` directory. The Manual Path preference determines where class and member descriptions are found. See “Preferences dialog” on page 232 for more information on preferences.

To load the Taligent project:


- 1 From the Project Editor File menu, choose “Open Project...” and select `Taligent.proj`.
- 2 Double-click on `Audio.h` to display the file in the Source Editor.

Opening the Documentation Browser

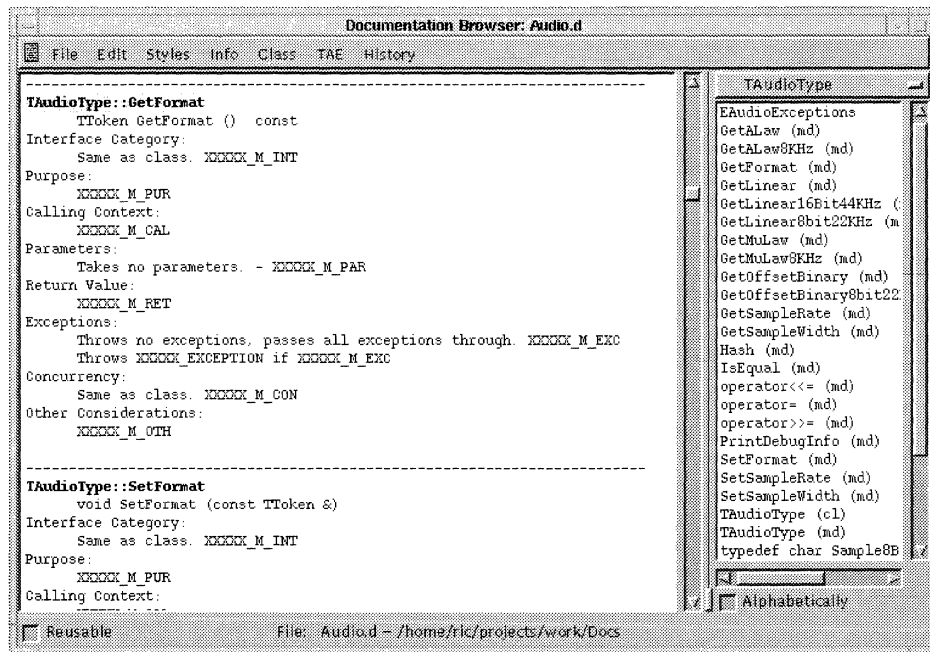
The Documentation Browser is similar to the Source Editor, but you view and work on the associated `.d` files.

- 1 In the Source Editor, select `GetFormat (md)` from the class list.
- 2 Choose “Show Documentation of `GetFormat`” from the Info menu.

The Documentation Browser displays the `Audio.d` file and the description of `GetFormat`.

 **NOTE** You can also start the Documentation Browser without a file displayed by selecting Documentation Browser from the Icon menu.

Icon menu reflects editing state
- read-only
- not modified
- modified
currently:
modified



Class pop-up menu either shows all classes or only one class (current setting)

Symbol list defined by class pop-up menu (clicking on a symbol positions the cursor)

Tool is reusable

Displays list alphabetically or in order of appearance in the file

Viewing other descriptions

As with the Source Editor, a list of classes appears at the right. You can display all classes or view only one class.

In addition, you can list the classes in alphabetical order or in the order they occur in the .d file by toggling the Alphabetically button at the bottom of the list. Click on any item in the list to view the description.

Changing from read-only to writable

If the file is read-only, you can view the documentation, but you can't change it, and obsoleted descriptions are not displayed. Check the icon to see if the file is writable. If you want to edit a read-only file, change the Preferences.

- 1 Choose "Preferences" from the Icon menu.
- 2 Press the button on the "Read-Only Documentation" flag to allow you to edit the file.


The icon on the Documentation Browser changes to indicate the file is writable and obsoleted descriptions are displayed.
- 3 Select OK to close the window.

Editing the file

You can select and type in the Documentation Browser the same way you do in the Source Editor. The Edit menu allows you to undo, redo typing, cut, copy, and paste.

You can emphasize text or change it back to default font.

Checking in and checking out files

 NOTE Checkin and Checkout functions are disabled in this release.


ADDING TALIGENT PUBLIC INCLUDES TO A NEW PROJECT

When you create your own projects in the Taligent Application Environment, you need to add Taligent public includes.

- 1 Open the Project Editor. Make sure the name of your new project is highlighted.
- 2 Choose “Load Subproject...” from the Project menu.
Now you see the File dialog prompting you for the project file.
- 3 Select `$TaligentSystemDocs/TaligentIncludesDocs/Taligent.proj` and commit.


COMPILING

SNIFF+ delegates compiling to a compiler of choice and interprets its output messages to allow fast positioning in the source code. With the product package, we supply the GNU gcc compiling system. Unless your system administrator installed SNIFF+ with another compiler, gcc will be called now. Make sure you are working with `filebrowser.proj`.

 NOTE If gcc is not installed on your system, you should skip this section.

Modifying the BrowserView class

- 1 Load class `BrowserView` into an Editor.
- 2 Position to the implementation of `BrowserView::BrowserView`.
- 3 Find the line where a new `ActionButton` is assigned to `shiftRight`.
- 4 Change the Label of the `ActionButton` from `>>` to `Down`

 NOTE Immediately after the code is changed, the tool icon at the upper left corner changes to the modified sign.

- 5 Insert an error by removing the comma `,` before the `Down`.

```
shiftLeft = new ActionButton(cIdShiftLeft, "<<");  
shiftRight= new ActionButto (cIdShiftRight, "Down");
```

- 6 Save the file by choosing `Save` from the File menu.
The tool icon changes back to its unmodified position.

Starting the compiler

Compiling BrowserView.o

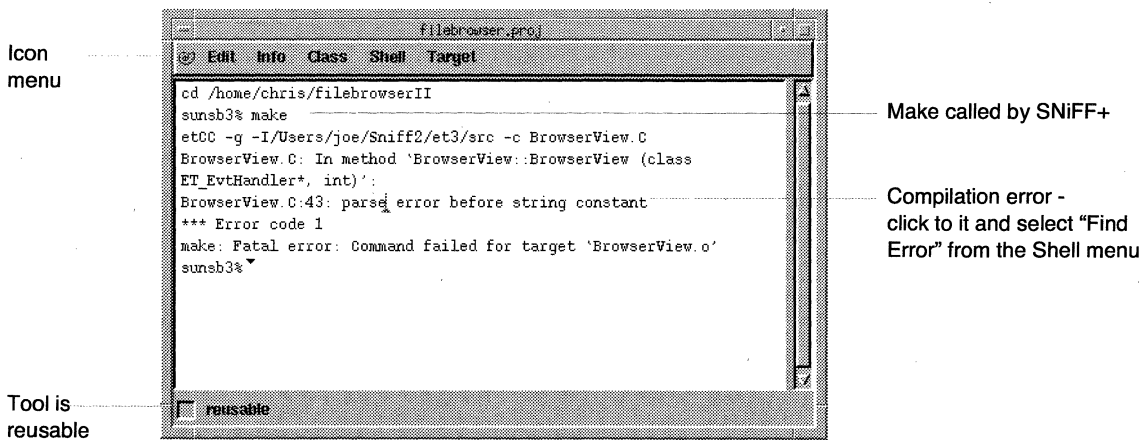
- Choose “Make File BrowserView.o” from the Make menu.

SNIFF+ now opens a Shell and starts the compiler with BrowserView.C. Since you entered erroneous code, the compiler outputs an error.

Jumping to the error in the source code and correcting it

- 1 In the Shell click in the line where the error is reported.
- 2 Choose “Find Error” from the Shell Menu.

The Editor positions the cursor to the line containing the error.



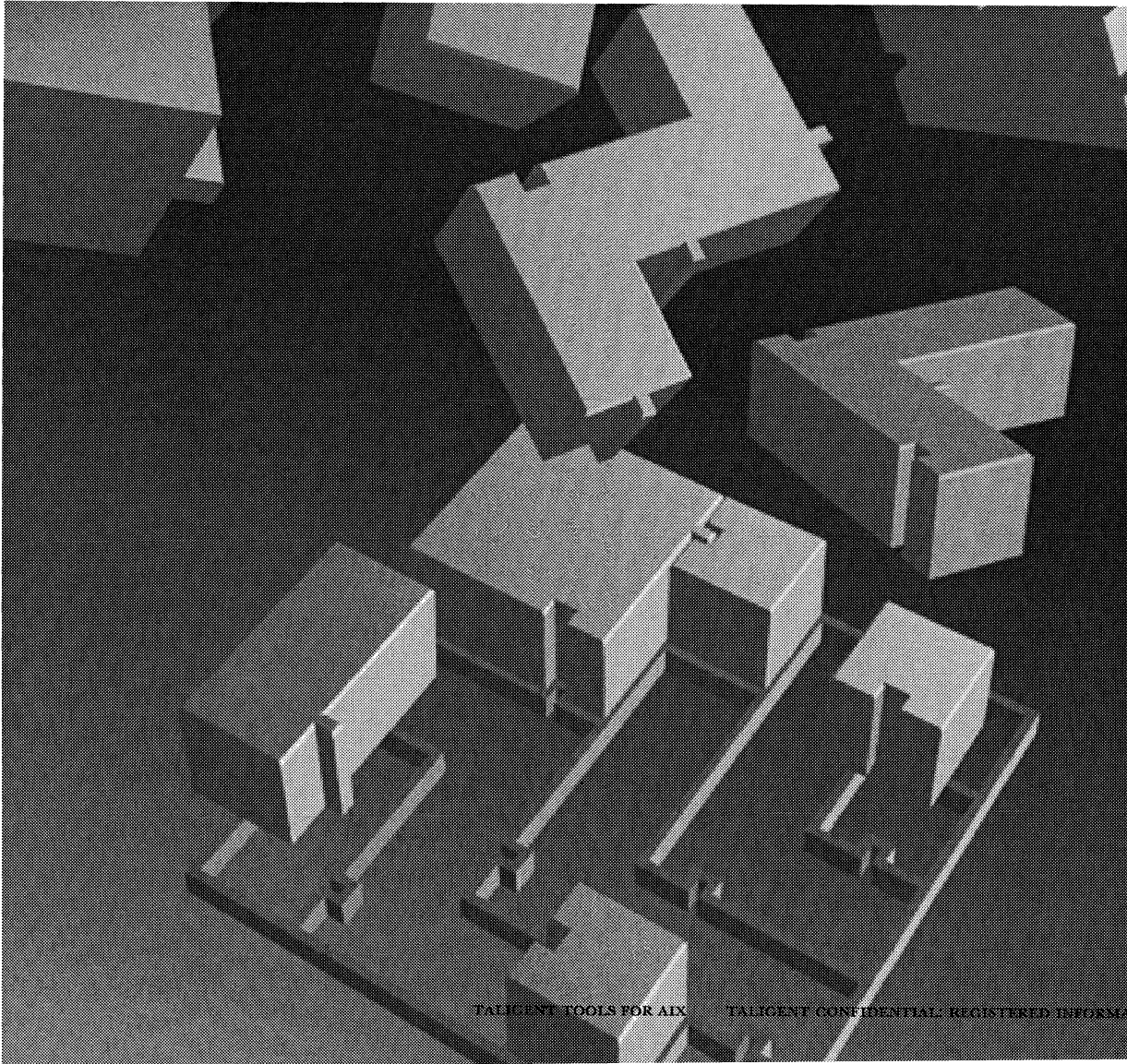
- 3 Correct the error by inserting the comma ',' before the “Down”.
- 4 Save the file.

Building the target executable

- Choose “Make Target filebrowser” from the Make menu (if this entry is disabled, you have forgotten to enter the target name in the Project Attributes dialog; see “Setting the project attributes” on page 138).

Now make is called, the modified source file is compiled, and the target is linked

NOTE In order to link the target, the correct target name must have been specified in the attributes of the filebrowser project (see “Setting the project attributes” on page 138).



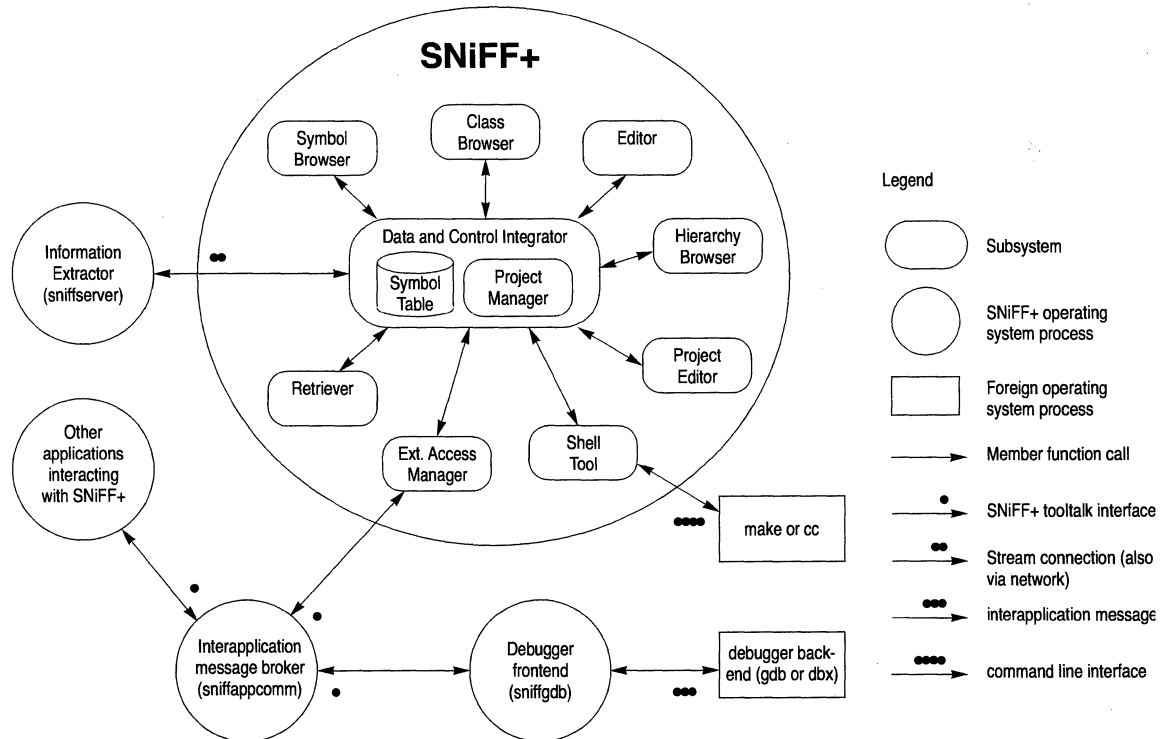
CHAPTER 12

BASIC ELEMENTS

SNIFF+ ARCHITECTURE

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.

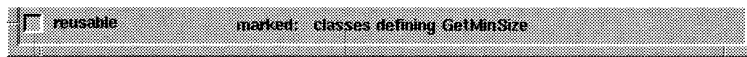


BASIC USER-INTERFACE COMPONENTS

SNiFF+ provides eight tools. These tools have different purposes, but they share a lot of functionality in several pull-down menus and the status line. This section starts with a description of the commonalities.


Status line

All SNiFF+ tools have a similar status line at the bottom which displays status information. Status information can be either a boolean value represented by a toggle button followed by a text, and/or a nonmanipulable text showing some information.



Toggle button determines the reusability state of the tool Status text; is not editable

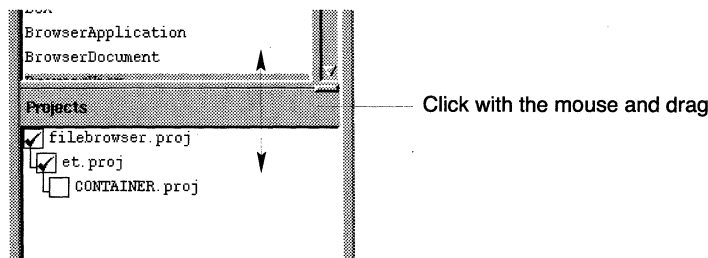
The reusable toggle button determines whether the tool can be reused in the case of a request or when a new tool has to be opened.

 **NOTE** It is good “SNiFF+ing style” to work with as few (reusable) windows as possible. This habit prevents screen and information cluttering.

Tool-specific status information is described in the corresponding tool sections.

Layout handle

All SNiFF+ tools consisting of more than one view have a layout handle. The layout handle allows modification of the size ratio between two views. By dragging the handle with the mouse, the ratio can be changed.

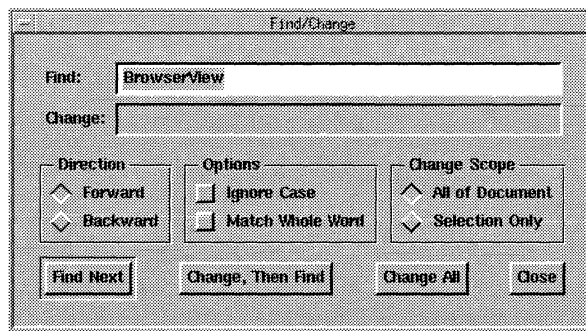


COMMON DIALOGS AND WINDOWS

Common dialogs and windows can be accessed from more than one tool in SNIFF+.

Find Dialog

The Find/Change dialog is accessible from tools containing text views (like the Editor and the Shell) via the “Find/Change...” entry in the Positioning and Edit menus. It allows finding and changing with regular expressions (see Appendix B). If the text is read only, a Find dialog is opened that does not allow changing text.



Text fields

Find	Describes the text that is to be found. It may contain regular expressions.
Change	Is the text that replaces a match on a change command.

Direction

Forward/ Backward	Is the search direction. The start of search is always the current cursor position.
----------------------	---

Options

Ignore Case	Specifies either a case sensitive or a case insensitive search.
-------------	---

Change Scope

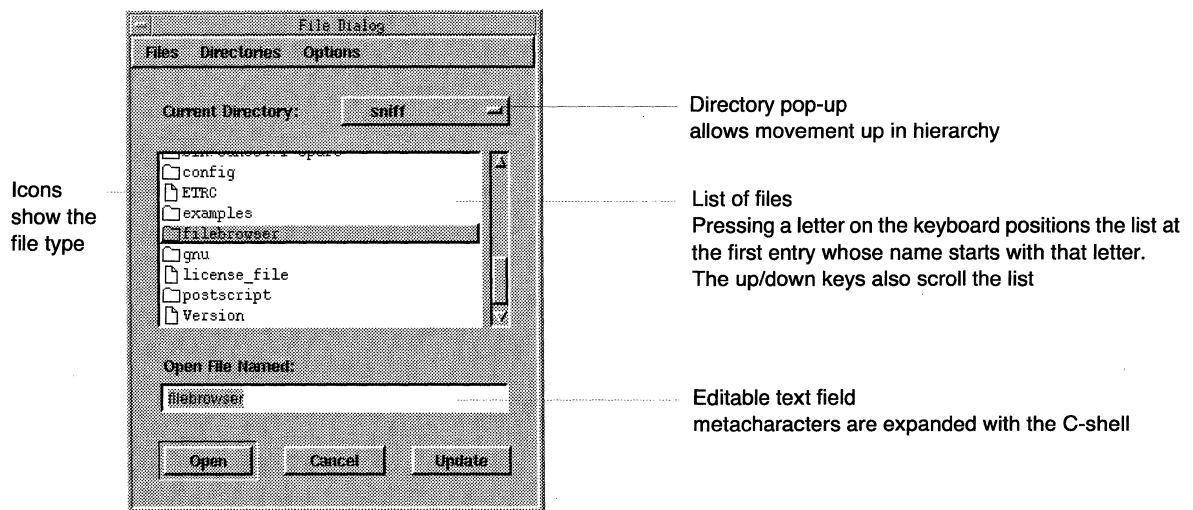
Match Whole Word	Specifies whether the search string must match a whole word. The default is that the search string is not restricted to being a whole word.
All of Document/ Selection	Only specifies whether the scope of the search is the whole document or the currently active selection (default is always the whole document).

Buttons

Find Next	Triggers the search for the next match.
Change, Then Find	Replaces the current selection with the change string, then starts a new search.
Change All	Changes all occurrences of the find string in the current change scope to the text entered in the change field.
Close	Closes the Find/Change dialog.

File Dialog

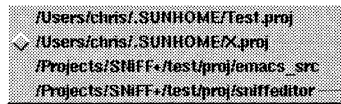
The File dialog is opened on save, new, and open file operations.



The text field expands C-shell metacharacters like '~' and \$variables. Pressing <Enter> in the text field selects the Open button.

Files menu

The Files menu lists the most recently opened files. Choosing a file from the list performs an open of the selected file and closes the File dialog.

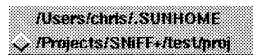


Permanent entry (retains until explicitly removed)

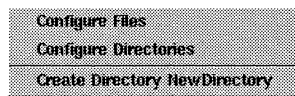
Normal entry

Directories menu

The Directories menu lists the most recently active directories. Choosing an entry from the list updates the directory.

**Options menu**

The Options menu serves to configure entries for the Files and Directories menu and allows creation of a new directory.



Configure Files	Opens a new dialog that allows making entries in the Files menu permanent. Permanent entries stay there all the time, regardless of how often they are selected.
Configure Directories	See “Configure Files” above.
Create Directory directory	Creates <i>directory</i> in the current directory. This entry is only enabled if the name of the new directory is entered in the editable text field.

Directory pop-up

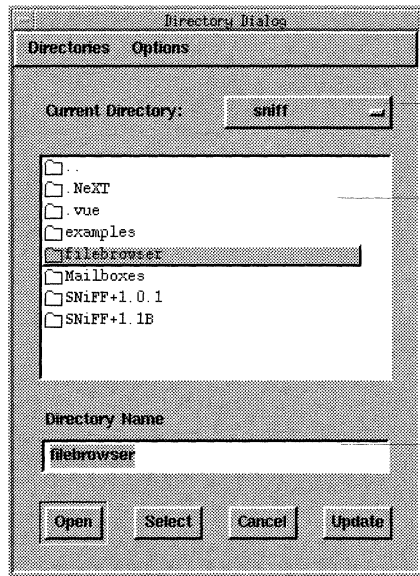
The Directory pop-up shows the parent directories of the current directory. Clicking on it allows fast navigation in the directory tree.

Buttons

Open	Opens the selected file and closes the File dialog.
Cancel	Closes the File dialog without any further action.
Update	Updates the file list (which is useful when new files are created or deleted while the File dialog is open).

Directory Dialog

The Directory dialog is very similar to the File dialog, but allows the selection of a directory rather than a file.



Directory pop-up
allows movement up in hierarchy

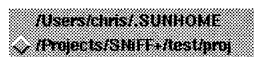
List of directories
Pressing a letter on the keyboard positions the list at
the first entry whose name starts with that letter.
The up/down keys also scroll the list.

Editable text field
metacharacters are expanded with the C shell

The text field expands C-shell metacharacters like '~' and \$variables. Pressing <Enter> in the text field automatically selects the Open button.

Directories menu

The Directories menu lists the most recently active directories. Choosing an entry from the list selects that directory and closes the Directory dialog.



Options menu

The Options menu serves to configure entries for the Directories menu and allows creation of a new directory.



Configure

Opens a new dialog that allows making entries in the Directories menu permanent. Permanent entries stay there all the time, regardless of how often they are selected.

Create Directory *directory*

Creates *directory* in the current directory. This entry is only enabled if the name of the new directory is entered in the editable text field.

Directory pop-up

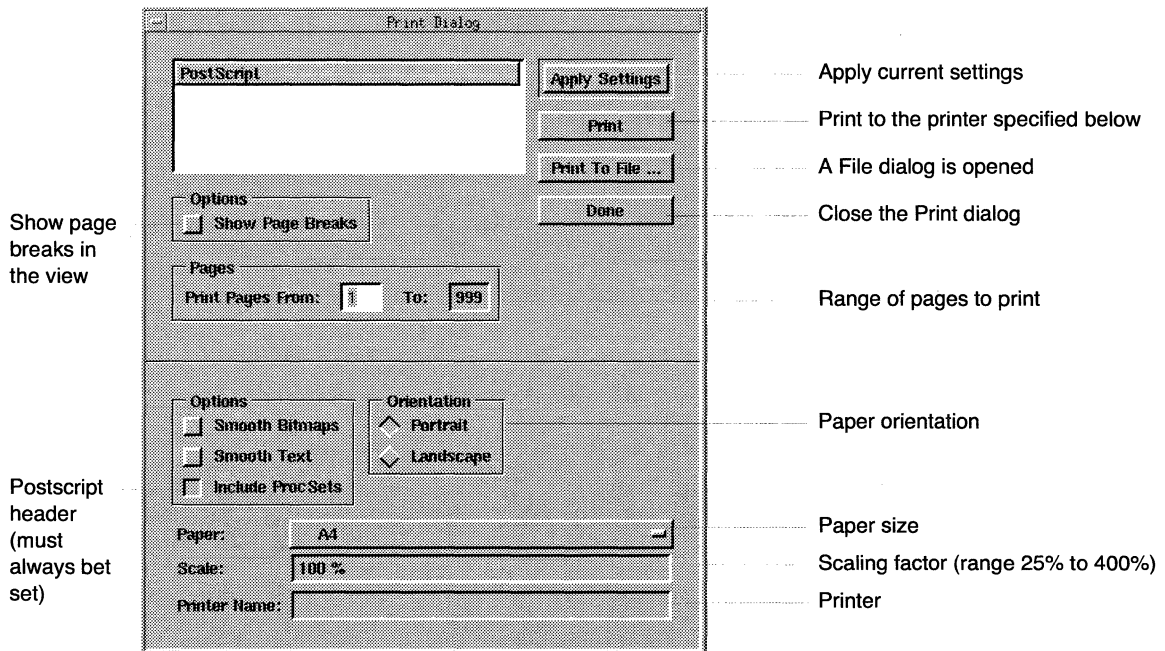
The Directory pop-up shows the parent directories of the current directory. Clicking on it allows fast navigation in the directory tree.

Buttons

Open	opens the selected directory and displays its contents in the directory list
Select	chooses the selected directory and closes the Directory dialog.
Cancel	closes the Directory dialog without any further action.
Update	updates the file list (which is useful when new files are created or deleted while the Directory dialog is open).

Print Dialog

The Print dialog is opened on print requests from the Hierarchy Browser and the Editor. It allows specification of printing options.

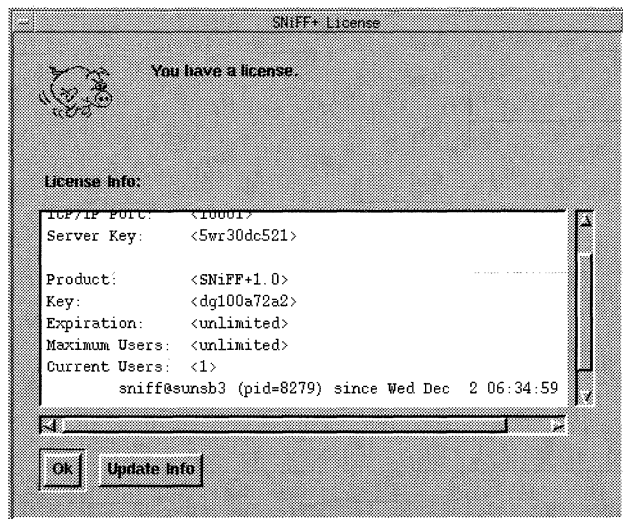


About dialog

The About dialog shows the version number of SNIFF+, copyright information, credits, and how to reach the authors.

License dialog

The License dialog displays information about the floating license server. The dialog can be opened by choosing “Licenses...” from the Icon menu. The dialog is automatically opened when there is a problem connecting to the license server process. A license is only allocated when a project is open.



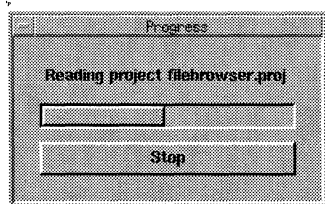
Area where reason for a problem is printed

License info view shows information about the license server and the currently active licenses

Updates the license info view

Progress Window

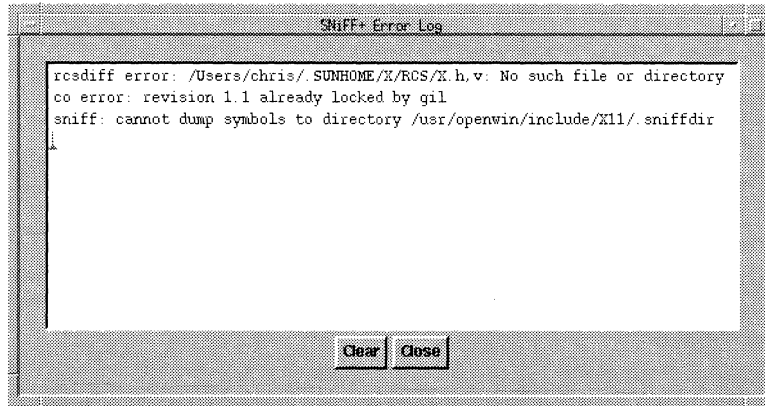
The Progress window appears whenever SNiFF+ needs some time to complete an operation. Examples are loading and closing of projects and retrieving a string in the Retriever.



Pressing the Stop button opens a dialog that allows stopping of the running operation. Some operations are not cancelable.

Error log window

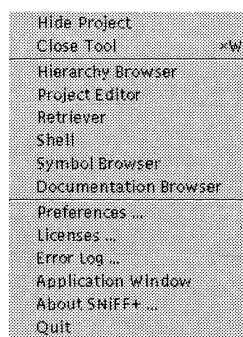
The Error log window displays SNIFF+ error and control messages. No messages are printed to the terminal where `sniff` is started. The window can be opened by choosing “Error Log...” from the Icon menu.



COMMON MENUS

Icon menu

The Icon menu groups together frequently used commands that have global character.



Hide Project	Hides the windows of all tools belonging to the corresponding project. This is useful to avoid screen cluttering while working with more than one project or with several tools. A hidden project can be shown with the Workspace Manager (see “Workspace manager” on page 185).
Close Tool	Closes the corresponding tool.
Hierarchy Browser	Brings a reusable Hierarchy Browser to the top of the display or opens a new tool if no reusable Hierarchy Browser is available.
Project Editor	Brings a reusable Project Editor to the top of the display or opens a new tool if no reusable Project Editor is available.
Documentation Browser	Brings a reusable Documentation Browser to the top of the display or opens a new tool if no reusable Documentation Browser is available.
Retriever	Brings a reusable Retriever to the top of the display or opens a new tool if no reusable Retriever is available.
Shell	Brings a reusable Shell to the top of the display or opens a new tool if no reusable Shell is available.
Symbol Browser	Brings a reusable Symbol Browser to the top of the display or opens a new tool if no reusable Symbol Browser is available.
Preferences...	Opens the Preferences dialog to edit user-specific settings (see “Preferences” on page 231).
Licenses...	Opens the License dialog, which shows information about the current license status of SNIFF+.
Error Log...	Opens the Error log window, which shows all SNIFF+ errors and other logging messages (see “Error log window” on page 177).
Application Window	Brings the Workspace Manager to the top of the display. This command is useful when the Workspace Manager is hidden below other windows.
About SNIFF+...	Opens the About dialog, which gives information about copyrights and how to reach the authors.
Quit	Terminates the current SNIFF+ session.

Info menu

The Info menu groups together commands for obtaining information about the current selection.

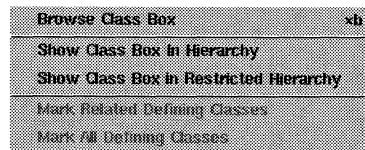
Edit Definition	⌘E
Edit Implementation	
Retrieve "About"	⌘Z
Retrieve "About" From Current Project	
Retrieve "About" From All Projects	
Find Symbols Matching "About"	⌘I
Find Symbols Containing "About"	
Copy Selected String	⌘C
Show Documentation Of	
Mark Documented	

Edit Definition	Loads the definition of the selected symbol (e.g., a class or an enumeration) into an Editor. The mouse shortcut for this command is a double click on the symbol.
Edit Implementation	Loads the implementation of the selected method into an Editor. This command is only enabled if a method is selected for which an implementation exists.
Retrieve selection	Opens a Retriever and retrieves all occurrences of <i>selection</i> from the currently selected projects only (see "Retriever" on page 210).
Retrieve selection From Current Project	Opens a Retriever and retrieves all occurrences of <i>selection</i> from the root project only (see "Retriever" on page 210).
Retrieve selection From All Projects	Opens a Retriever and retrieves all occurrences of <i>selection</i> from all projects (see "Retriever" on page 210).
Find Symbols Matching selection or	

Find Symbols Containing selection	Serves to get information about all symbols with the current selection as name or as part of the name. Both commands obtain a Symbol Browser and start the corresponding query (see “Symbol browser” on page 204).
Copy Selected String	Is enabled for browsing tools. It corresponds to the Copy command of text-based tools and copies the string of the selection to the clipboard.
Show Documentation of selection	Obtains a Documentation Browser with the documentation of the selected symbol (see “Documentation Browser” on page 224). This entry is enabled only if there is documentation for <i>selection</i> .

Class menu

The Class menu serves to issue commands for obtaining further class-specific information about the current selection (the entries are only enabled if the selection is a class).



Browse Class class	Loads <i>class</i> into a Class Browser (see “Class browser” on page 206).
Show class in Hierarchy or Show class in Restricted Hierarchy	Obtains a Hierarchy Browser and loads either the entire class graph or the graph of the base and derived classes. The selected class is highlighted in the Hierarchy Browser (see “Hierarchy browser” on page 208).
Mark Classes Defining method or Mark Related Classes Defining <i>method</i>	Obtains a Hierarchy Browser and loads either the entire class graph or the graph consisting of the selected class and its base and derived classes. All classes defining <i>method</i> are marked in the Hierarchy Browser (see “Hierarchy browser” on page 208).

Filter menu

There are two possibilities to restrict the amount of information in SNIFF+ tools that display information in a list (i.e., the Symbol Browser, the Class Browser, and the Retriever).

Set Filter...	x f
Reset Filter	x g
Select From "EvtHandler" Only	x o
Select From All Classes	x a

It is possible to define a regular expression which filters the list via the "Set Filter..." command.

Set Filter... Opens a filter panel that prompts for a regular expression filter (see "GNU Regular Expressions" on page 257).

Reset Filter Resets the filter to allow all entries.

The list can be further restricted by means of the contents of the view at the bottom, which shows either the inheritance graph in the case of a Class Browser or the project tree in all other cases. If a project/class has a checked checkbox in front of it, its corresponding information is displayed in the list.

Which information is displayed can be determined either by clicking on the checkboxes directly or by setting them via the Filter menu.

Select From class/project Only displays entries in the list belonging to the *class/project* selected in the project tree.

Select From All Classes/Projects Displays entries from all classes/projects.

History menu

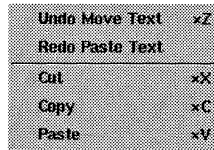
The History menu serves to reset the tool to a previous state (or to issue an earlier query again). The structure of the entries depends on the kind of tool from which the menu is invoked.

class:Cluster type:instance variable
class:Cluster type:method
class:Box type:method
class:Button type:method


The number of queries to be remembered can be specified in the Preferences dialog (see "Preferences" on page 231).

Edit pop-up menu

The Edit pop-up menu appears when the right mouse button is pressed in any editable text item or in a text view.



Undo command	Undoes the last change (<i>command</i>) to the text. The number of remembered undoable commands can be specified in the preferences file (see “ETRC file entries” on page 261).
Redo command	Redoes the last undone change (<i>command</i>).
Cut	Cuts out the current selection into the paste buffer. The entry is only enabled if there is an active selection.
Copy	Copies the current selection into the paste buffer. The entry is only enabled if there is an active selection.
Paste	Pastes the contents of the paste buffer at the current cursor location. If there is an active selection, the selection is replaced by the pasted text. The entry is only enabled if the paste buffer is not empty.

 **NOTE** The Undo, Redo, Cut, Copy, and Paste commands are also accessible from the Edit pull-down menu in tools like the Editor and the Shell.

SHORTCUTS

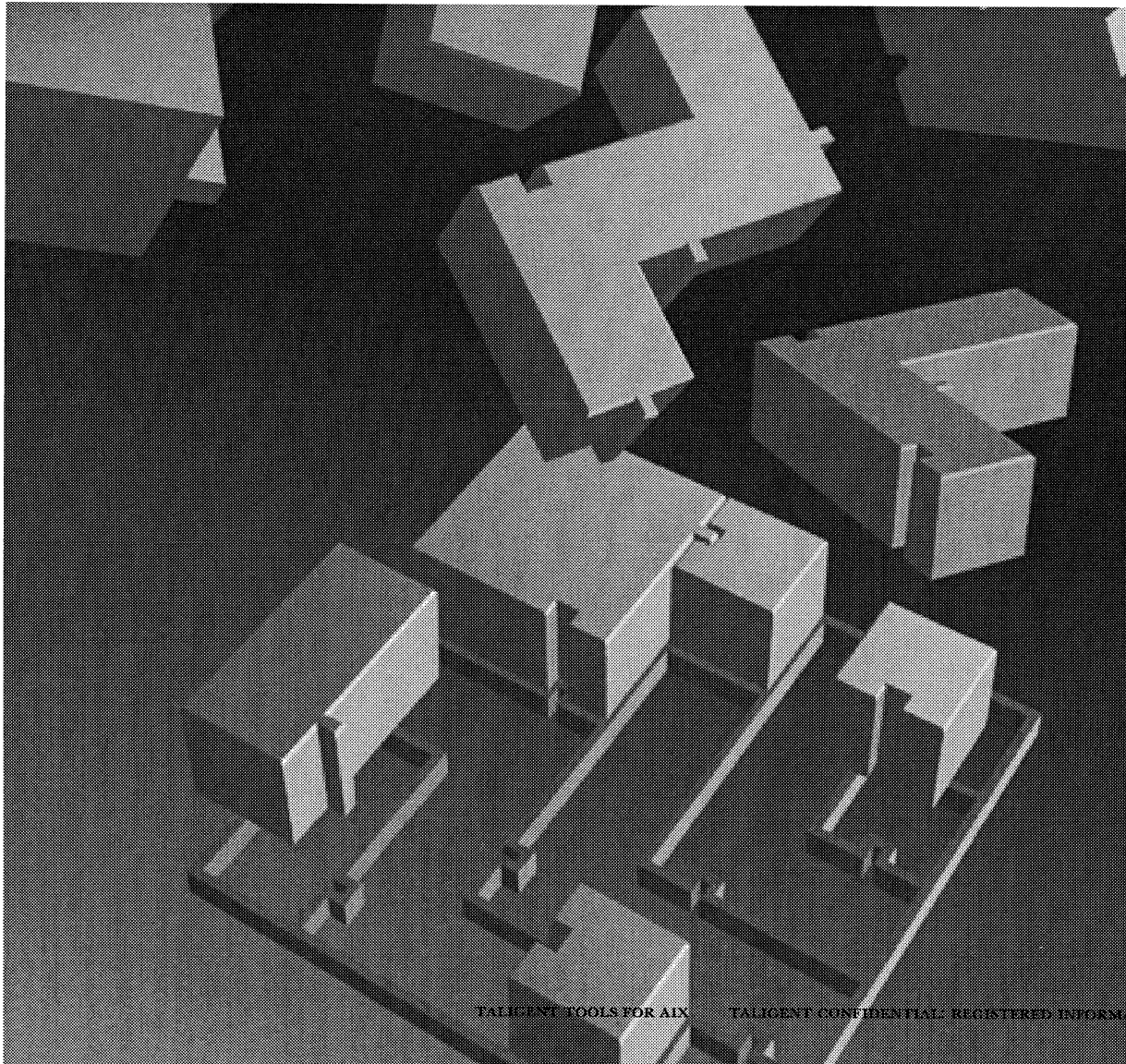
Keyboard shortcuts

The complete functionality of SNIFF+ is provided via the menus of the various tools. To speed up the work, especially for experienced users, SNIFF+ also provides three different types of shortcuts to allow faster access to the commands found in menus.

- 1 Keyboard shortcuts are issued by holding down <Alt> of your keyboard and pressing the key that is shown at the right of a menu entry. Some frequently used shortcuts are:
 - ⌘ <Alt>C for copy
 - ⌘ <Alt>V for paste
 - ⌘ <Alt>B for browse class
- 2 Mouse shortcuts are issued by double-clicking with the mouse on entries in lists or on selectable items. Some frequently used shortcuts are:
 - ⌘ Editing the source of a symbol by double-clicking on it in the Symbol Browser
 - ⌘ Jumping to the source location of a variable by double-clicking on it in the Class Browser
- 3 Deep clicks are issued by holding down the <Ctrl> key and pressing the left mouse button. Some frequently used deep clicks are:
 - ⌘ Switching from the declaration of a symbol to its implementation by <Ctrl>clicking on the symbol in the symbol list of the Editor
 - ⌘ Restricting the information shown in the list of a Symbol Browser by <Ctrl>clicking on the checkbox of a project in the project tree view
 - ⌘ Showing methods of only one class in the Class Browser by <Ctrl>clicking on the class in the inheritance graph view

Fast positioning in lists

Pressing an alphabetical key while the mouse pointer resides over a list will position the list to the first entry whose name starts with that letter.

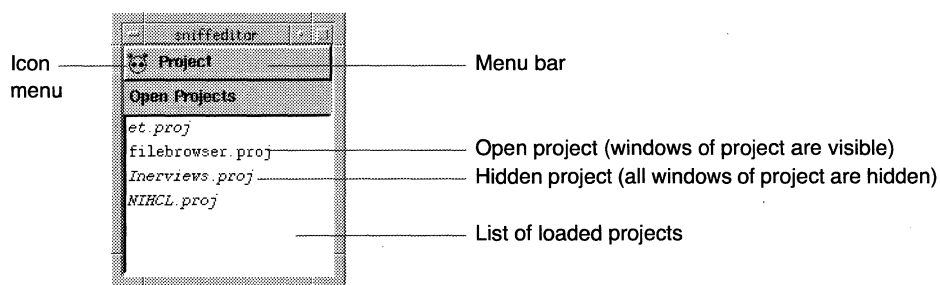


CHAPTER 13

SNIFF+ SUBSYSTEMS

WORKSPACE MANAGER

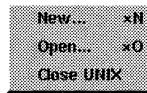
The Workspace Manager serves to manage projects and open tools. It consists of a menu bar and a list of open projects.


Commands in the list of loaded projects


- | | |
|-----------------|--|
| Double-click | On a loaded project hides/shows all windows belonging to that project. This command is also available via the Icon menu. |
| <Ctrl>
click | On a project shows the windows of that project and hides the windows of all the other projects. |

Project menu

The Project menu serves to create new, open existing, and close open projects, as well as to quit SNIFF+. For a description of project files see “Project file” on page 239.

**New...**

Opens a Directory dialog, which prompts for the directory where the source files of the new project are located. After the directory has been specified, a Project Editor is opened asking whether to load all existing C/C++ files. The newly created project has to be stored in a project file.

 **NOTE** To enhance the transportability of a project, the specification of the source files directory can contain environment variables. Selecting directories with the browser of the Directory dialog always stores the absolute directory path into the project file. Entering the complete directory specification in the text field at once retains used environmental variables and other shell metacharacters like '~'. To change the source path of a project after it is created, see “Project Attributes Dialog” on page 199.

Open...

Opens a File dialog, which prompts for an existing project file. After a project file has been specified, the project is loaded into SNIFF+ and the environment (all window positions, sizes and contents) is restored to the same status as when the project was closed the last time.


Close project

Closes the selected *project* and all windows belonging to it. If the structure of the project has been modified, a panel is opened asking whether the project file should be saved. On a reopen, the project environment is restored to the same status as when the project was closed.

PROJECT EDITOR

The Project Editor is used to edit and browse project-specific information:

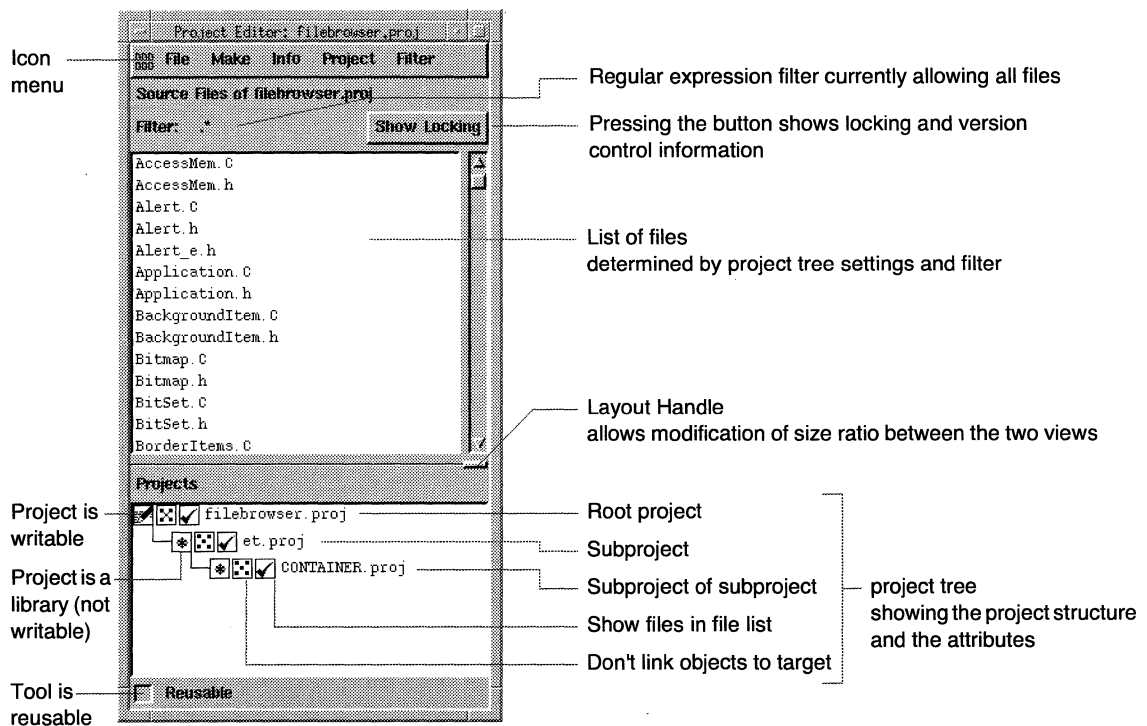
- ✦ Project attributes
- ✦ Information about which files belong to the project
- ✦ Subprojects
- ✦ Version control and locking information.

 **NOTE** While the Project Editor serves to browse the entire tree of projects that are loaded, only the structure of the root project can be edited. To change the structure of a subproject, it has to be opened as a root project itself.

The Project Editor stores the project information in project files (see “SNIFF+ projects” on page 244).

A Project Editor can be opened by choosing “Project Editor” from the Icon menu.

A Project Editor contains two views. The upper view displays a list of files and the lower view shows the project tree. The amount of information shown in the list of files can be controlled with the Filter menu (see “Filter menu” on page 181).



File list

The list of files shows the files belonging to the root project and its subprojects. The list is restricted by the setting of the project tree (checkboxes) and by the filter.

Project tree

The project tree shows the hierarchical project structure including all subprojects. It also shows whether the project is writable and whether its objects should be linked to the project target of the root project. The attributes displayed in the project tree can be edited in the Project Attributes dialog (see below).



Project is a library; project attributes may be modified but not the source files



Project is read-only; neither attributes nor source files may be modified



Project is writable



Link objects of Project to target



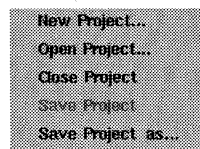
Don't link objects of Project to target

The checkbox of the project defines whether the files of the project are shown in the file list. They can be modified directly with the mouse or can be set by the Filter menu.

A deep click (<Ctrl>click) on a project entry (not on the checkbox) selects only files from that project and hides all others.

File menu

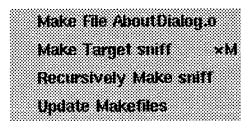
The Project Editor's File menu serves to issue commands that create, open, and save projects, as well as to quit SNIFF+. For a description of project files see "Project file" on page 239.



New Project...	
Open Project...	
Close Project	The same entries as in the Project menu of the Workspace Manager.
Save Project	Saves the project file (this entry is only enabled if the project structure or attributes have been modified since the last save).
Save Project As...	Opens a File dialog, which prompts for the new project file name.

Make menu

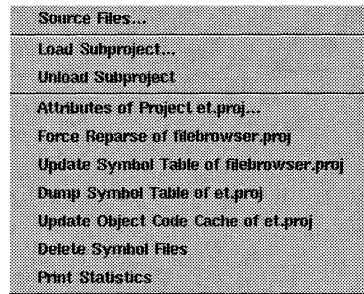
The make menu serves to issue make commands. The command actually issued by these can be specified in the Preferences dialog and in the Project Attributes dialog (see “Project Editor” on page 187).



Make File objectfile	Obtains a Shell and starts “make <i>objectfile</i> ” in the project's source directory.
Make target target or Make Project <i>project</i>	Obtains a Shell and starts “make <i>target</i> ” or “make <i>project</i> ”, respectively, depending on whether the selected <i>project</i> has a defined target, in the <i>project's</i> source directory. If no project is selected in the project tree, the root project is taken.
Recursively Make <i>target</i> or Make All Writable Projects	Obtains a Shell and starts the “make” command for all writable subprojects. Finally, “make <i>target</i> ” (or only “make” if the root project doesn't have a defined target) is called for the current root project.
Update Makefiles	Updates the dependency information for the makefiles of all writable projects. This command has to be issued only when a new include statement is inserted in one of the source files of the project. This command need not be issued when attributes or the project structure are changed, in which case SNIFF+ updates the makefile information automatically.

Project menu

The Project menu serves to issue commands that manipulate the attributes and structure of the current project. It is not possible to change information defined in a subproject. To change a subproject, it has to be opened as a root project on its own.



- | | |
|--|--|
| Source Files... | Opens the Source Files dialog that serves to define the project's source files (see "Source Files dialog" on page 197). |
| Load Subproject... | Pops up a File dialog box which prompts for the project file of a subproject to be added. It is only possible to load subprojects for the root project. If a subproject is to be added to a subproject, it has to be opened as a root project on its own. |
| Unload Subproject | Purges the selected subproject from the current project. This command is only enabled for subprojects of the root project. |
| Attributes of Project <i>project</i> ... | Opens the Project Attributes dialog described on page 199. |
| Force Reparse of <i>project</i> | Performs a reparse of the selected <i>project</i> . A reparse is necessary, e.g., if the parser configuration file has been modified (see "Dealing with preprocessor macros" on page 236). |
| Update Symbol Table of <i>project</i> | Checks for all files belonging to <i>project</i> if the corresponding source files were changed since the symbols were loaded and reparses only the modified files. This command has to be executed only if project files were modified with tools other than SNIFF+ (see "Editor" on page 213). |

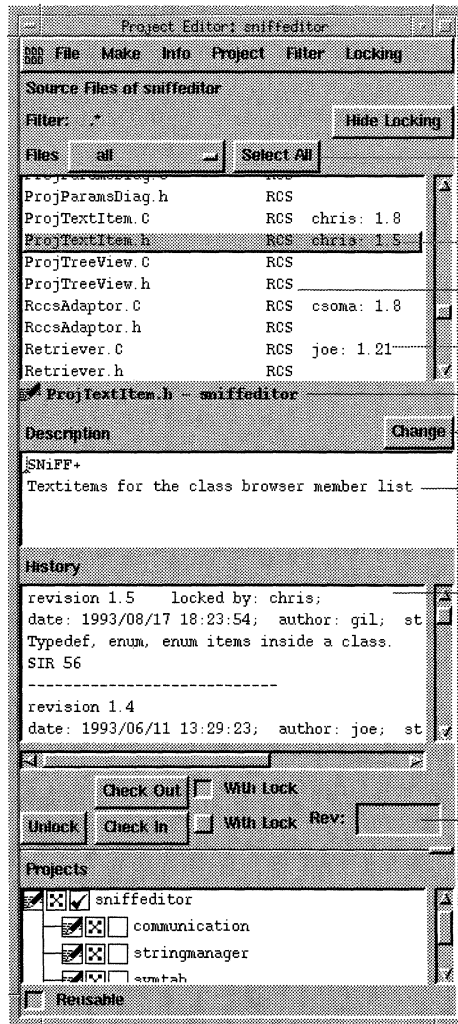
Dump/Remove Symbol Table of <i>project</i>	Dumps or removes a project-specific symbol table containing the symbolic information about the whole project. This entry is only enabled if the project is a library project. Further explications can be found in “Tuning and persistency of symbolic information” on page 243.
Delete Symbol files	Deletes all symbol files of this project. Symbol file management is normally fully transparent to the user. This command is necessary only if the symbol files have a wrong modification date (due to a copy or some other reason) or were corrupted. On a project close then, new symbol files will automatically be created.
Print Statistics	Displays the number of symbols, files, and included files for every project, as well as a summary of all projects, to a reusable Shell.

Show Locking button

Pressing the Show Locking button shows locking and version control information in the Project Editor.

Project Editor with locking information shown

When the Project Editor is opened for the first time, locking and version information is hidden. After the Show Locking button is pressed the Project Editor displays also locking and version information for the selected projects. For a general discussion of the integration of version control systems in SNIFF+, see "Version control" on page 254.



List allows multiple selections.
A selection can be extended by pressing <SHIFT> and selecting entries. All entries of a list can be selected by pressing the "Select All" button.

Type of the version control system

Locker(s) and locked version(s)

File status, source file name and project name

Descriptive text of the selected source file can be changed with the "Change" button

History of the selected source file

Revision number for check in, check out and unlock operations

File list and status line

The file list is a multiple selection list and displays the following information:

Name of the file	Is the same as in the Project Editor without the locking information.
Underlying version control system	Is the type of the version control system used. The following systems are currently supported: RCS, SCCS, SNIFF and none if no version control system is used. Different version control systems can be used for different projects. The Project Attributes dialog (see "Project Attributes Dialog" on page 199) determines which version control system is used.
User name of locker and locked version	Is only displayed if the file is locked by somebody.

The status line displays the state of the file, the filename and the project name it belongs to.

The modification state icon is determined by comparing the working file with the version file and can be one of these:



Working file is not writable and not modified



Working file is writable and not modified

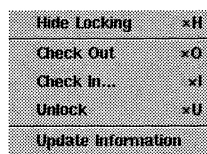


Working file and last checked in version differ. Either the working file is modified or a new version has been checked in by somebody else

The icon can be empty if the file has never been checked into the version system.

Locking menu

The Locking menu is only visible if the Project Editor shows locking information.



Hide Locking

Hides the locking information in the Project Editor and also hides this menu (same effect as pressing the Hide Locking button).

Check Out

Checks out the selected file(s). If a valid version number is entered in the Revision text field (see below), this version is checked out, else the latest version is checked out. If multiple files are selected, the latest versions are checked out. Depending on the “with lock” button setting, the locking of the file is influenced in the following way:

- With lock (default) the file(s) are checked out writable and locked for the current user.
- No lock the file(s) are checked out read only and are not locked.

The “Check Out” menu entry in the SNIFF+ Editor always checks out the file with a lock (see “File menu” on page 188).

Check In...

Checks in the currently selected file(s). A Log message dialog box is opened, prompting for the log message to be saved with the checked in version. The log messages for the various versions of a file are displayed in the History text.

If a valid version number is entered in the Revision text field (see below), this version is checked in, else a new version in the current branch is checked in. If multiple files are selected, the latest version of the currently locked branch is checked in. If a working file is not modified, a dialog asks whether the file should still be checked in. Depending on the “with lock” button setting, the locking of the file is influenced in the following way:

- With lock: the file(s) are checked in as new version(s) but are still locked for the current user and writable.

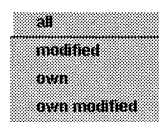
- ⌘ No lock (default): the file(s) are checked in, the lock of the current user is released, and the protections of the working source file(s) are set to read only.

The “Check In...” menu entry in the SNIFF+ Editor always checks in the file and removes the lock (see “File menu” on page 188).

Unlock	Removes the lock from the selected file(s) and sets the protection of the working source file(s) to read only. The version file(s) in the version control systems return to the same state they had before the lock was set. If the working file is modified, a dialog asks whether the lock should be removed. The Unlock entry is only enabled if the selected file is locked. A revision number can be entered to cover the case that several revisions are locked on different branches.
Update Information	Updates the information of all files displayed in the file list.

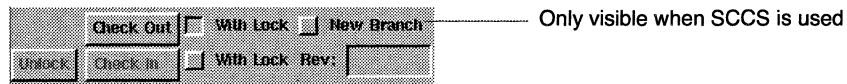
Files popup button

The files popup button is used to constrain the list of displayed files.



all	Displays all files regardless of the locking state.
modified	Displays only working files that are different compared to the last checked in version. There can be two reasons for that: <ul style="list-style-type: none"> ⌘ The working file is modified by the current user ⌘ A new version has been checked in by someone else and the working file is out of date.
own	Displays only files that are locked by the current user.
own modified	Displays only files that are locked by the current user and are modified.

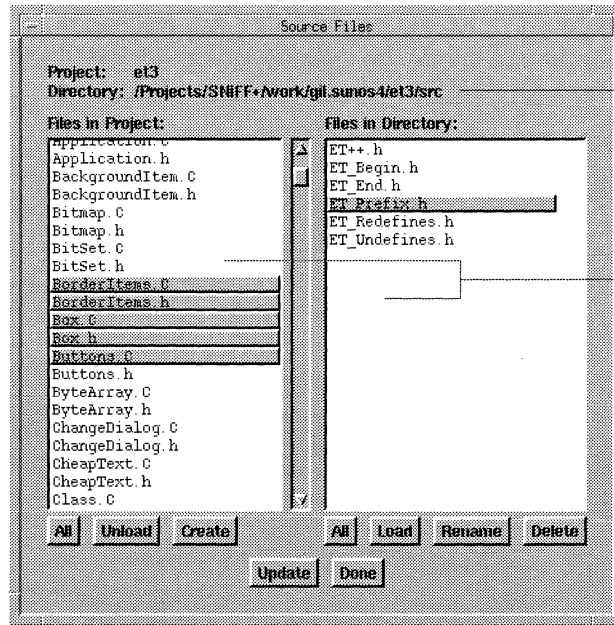
- Description text** The Description text displays the description for the selected file. It also serves to enter the description for a newly checked-in file. The text can be of arbitrary length and may contain new-line characters. The description text can be changed and stored with the “Change” button and with single-file check-ins.
- History text** The History text shows the log entries for the checked-in versions. Most recent entries are displayed at the top, old entries at the bottom. RCS symbolic names are displayed at the very bottom.
- Check Out button** The Check Out button has the same effect as choosing “Check Out” from the Locking menu.
- Check In button** The Check In button has the same effect as choosing “Check In...” from the Locking menu.
- Unlock button** The Unlock button has the same effect as choosing “Unlock...” from the Locking menu.
- Revision text field** This field is used for check in, check out, and unlock operations and determines what version of the selected file is checked in/out. If the field is empty, the latest version of the file is taken.
- SCCS only: New Branch button** When SCCS is the version control system, a new button appears above the Revision text field:



The New Branch button allows the creation of new branches during SCCS check-out operations.

Source Files dialog

The Source Files dialog serves to handle the source files of a project. It is opened by choosing “Source Files...” from the Project menu.



Project directory

List of files

with 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

Buttons

All	Selects all elements in the list.
Unload	Unloads the selected source file(s) from the current project. All symbols of an unloaded file are removed from the project. This command is only enabled for files of the root project. A file can also be unloaded by double-clicking.
Create	Pops up a dialog box, which prompts for the name of a source file to be created and loaded. The file name must have one of the legal extensions for include or implementation files. (Legal extensions can be specified with the Preferences dialog as described in “Preferences” on page 231.) If a legal filename was specified, no file with the indicated name exists, and access permission allows the creation of the file, then it is created and filled with a template. A template name starts with the string “template.” and has the same extension as the newly created file. User-specific templates can be provided by storing them in a directory that is specified in the Preferences dialog. Site-specific templates are stored in the config directory of the SNIFF+ installation directory (see “Preferences” on page 231). If no templates are provided, then an empty file is created.
Load	Loads the selected source file(s) into the project. To load a file into a project means to parse the file and load the symbolic information. A file can also be loaded by double-clicking.
Rename	Pops up a dialog box, which prompts for the new name of the file. Only files in the directory can be renamed. To rename the file of a project, unload the file, rename it and load it again. The button is only enabled if a file is selected.
Delete	Deletes the selected file(s) if file permission allows. SNIFF+ asks for confirmation before actually deleting the file(s).
Update	Updates the lists of file, which is necessary, e.g., if a new file is created in the shell.
Done	Closes the Source Files dialog.

Project Attributes Dialog

Project specific parameters are edited with the Project Attributes dialog and are stored in the project file. Some of the parameters override user preference settings (see “Preferences” on page 231).

Changes to the project attributes take immediate effect if not otherwise specified in the text below. Attributes of frozen projects cannot be changed.

The following picture shows the Project Attributes dialog for a root project.

The screenshot shows a dialog box titled "Attributes of Project f.proj". It contains several fields and checkboxes for configuring project attributes:

- Target:** filebrowser
- Source Path:** /Users/sniff/filebrowser
- Tab Width:** 8
- Generate Dir:** /Users/sniff/filebrowser/sniffdir
- Parser Config File:** %SNIFF_DIR/config/et_parser_config
- Class Prefix:** ET_
- Make Command:** make
- Makefile Support:**
- Overlay Files:**
- Locking Parameters:**
 - Tool:** RCS
 - Path:** /Users/sniff/filebrowser
- Project Parameters:**
 - Project is Library
 - Link Objects of Project

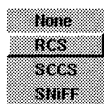
At the bottom of the dialog are "Ok" and "Cancel" buttons.

Target	Defines the name of the target of a project. The target name is used to drive the make command and the Debugger.
Source Path	<p>Specifies the directory where the source files of a project are located. In the standard case this path is set automatically when the project is created and is never changed.</p> <p>The reason for changing the Source Path attribute is to improve the transportability of a project. If a project has to be transported to another place in the UNIX directory tree, two problems occur. First, the source files cannot be found anymore because the absolute path name of the source directory is stored in the project file. Second, the project cannot be compiled because the path names of generated makefile parts are outdated. The first problem is solved by asking the user for the new source directory path when the source files cannot be found anymore in opening a project. The second problem can be solved by updating the make support files manually (see “Make menu” on page 189).</p> <p>A general way to enhance transportability of projects is to use environment variables as part of the source path specification. This also allows project team members to work on the same project but on different NFS mount points in a network.</p>
Tab Width	Is used to specify the length of the spacing between two tab stops. The default width is set in the Preferences dialog (see “Preferences” on page 231).
Generate Dir	Indicates the directory where SNIFF+ puts the project-specific files generated for this project (see “Files created and used by SNIFF+” on page 239). Per default a directory <code>.sniffdir</code> is created in the source directory of the project. You may want to change the directory if you are not allowed to create a subdirectory in the source directory of the project. SNIFF+ displays a warning message in the Error log if permissions prevent writing to the generate directory.

- Parser Config File** Indicates the file where special configurations for the information extractor are stored (see “Files created and used by SNIFF+” on page 239). If a new parser configuration file is specified, or an existing file has been changed, SNIFF+ has to reparse the source file(s) with the changed configuration. Reparsing can be forced by issuing “Force Reparse” from the Project menu. Effect of the change: Reparse of the project.
- Make Command** Specifies the command to be submitted to the Shell when a make command is issued (see “Make menu” on page 189). The default is set in the Preferences dialog (see “Preferences” on page 231). If you compile on a compile server, you can change the command, for example, to “on server make”, or you can provide your own shell script to do fancier things.
- Class Prefix** Is used only in conjunction with debugging. To enhance integrability with other software systems, class libraries sometimes add a prefix to classes with a macro. This prefix change is not visible to SNIFF+ since the information extractor does not do macro expansion. To allow transparent symbol matching between browsing and debugging, the class prefix attribute may be set. ET++, for instance, uses the class prefix 'ET_' for all of its classes.
- Makefile Support** Specifies whether the support files (`ofiles.incl` and `dependency.incl`) for the makefile are generated (see “Makefile Support” on page 249).
- Overlay Files** Specifies whether files of subprojects should be overlaid by files with the same name of superprojects. If this option is set in the sub- and superproject, SNIFF+ hides the symbols of files which are overlaid. This feature enhances the teamwork support (see “Teamwork Support” on page 234). If this option is switched off, all files are loaded, even if two of them have the same name. Default is not to overlay files. Change takes effect on the next project open.

Locking Parameters

Tool determines which version control system is used for the project.



A description of the integration of the different tools can be found in “Version control” on page 254. SNIFF locking is a simple SNIFF+ internal locking but without the version control features of the other supported tools.

Path Defines the path where SNIFF+ searches for the version tool subdirectories. The default path is the source directory of the project. The path specification must not contain the directory name of the version control system. The following directories are added to the *path* for the various version systems:

Version control system	Directory
RCS	RCS
SCCS	SCCS

If the directory of the version control system is not located in the source directory of the project, no link to the actual version control directory is needed. Just enter the path where the version control directory is located. Several SNIFF+ projects can share one common version control directory.

Project parameters

Project is Library	Specifies whether the project is a library project. Library projects are frozen and cannot be modified. This information is also shown in the project tree of the Project Editor (see “Project Editor” on page 187).
Link Objects of Project	Specifies whether object files of the project have to be linked to the target. This information is also shown in the project tree of the Project Editor (see “Project Editor” on page 187).

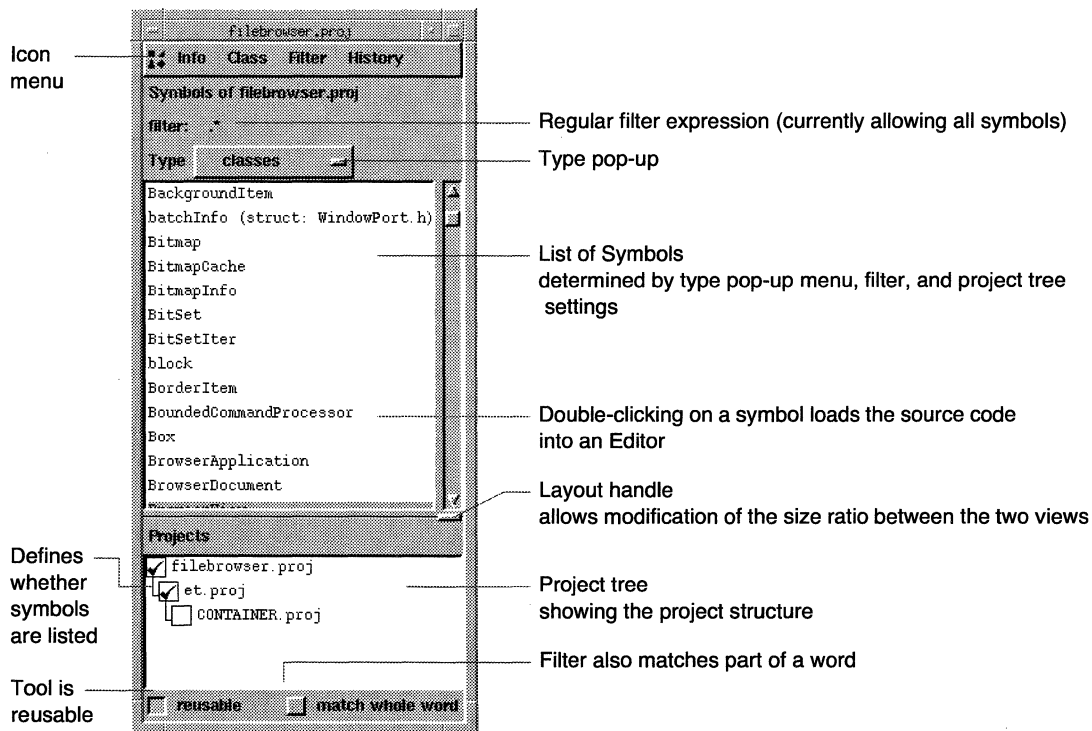
Project Attributes dialog for frozen subprojects

Attributes of frozen subprojects cannot be changed. Only the “Project Parameters” can be overridden if the overriding results in a restriction. E.g., an editable subproject can be frozen, but a frozen subproject cannot be turned editable. If project files are to be linked, this can be turned off, but not vice versa. If the attributes of a frozen subproject have to be changed the project must be opened as a root project and must be unfrozen.

SYMBOL BROWSER

The Symbol Browser consists of a list of symbols whose content is determined by the type pop-up menu, the project tree settings, and the filter field. The type pop-up menu allows selection among C++ constructs such as classes, methods, and variables, as well as preprocessor macros. The project tree shows the project structure and makes it possible to select the projects whose symbols are displayed.

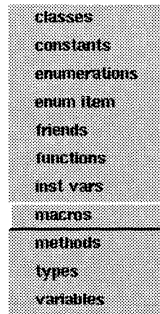
Symbol Browsers are obtained by issuing the “Symbol Browser” command in the Icon menu. The other way to obtain a Symbol Browser is to issue a “Find Symbols Matching *selection*” or a “Find Symbols Containing *selection*” command from the Info menu. In this case they show a symbol list that is filtered by *selection*.



Abstract classes (i.e. classes that define a pure virtual method) are displayed in italic font face.

Type pop-up

The type pop-up specifies what type of symbols are shown in the list.



C structures and unions as well as typedefs for structures and unions are listed as class types, whereby the names of these types are marked as “(struct)” or “(union)” in the symbol list. C++ templates are also listed as classes, whereby the names of the templates are followed by “(template)”.

The list of methods and instance variables can get very long, since they are flat views of all methods/variables in the project. Symbols of the same type with the same name are qualified by the name of the file they belong to.

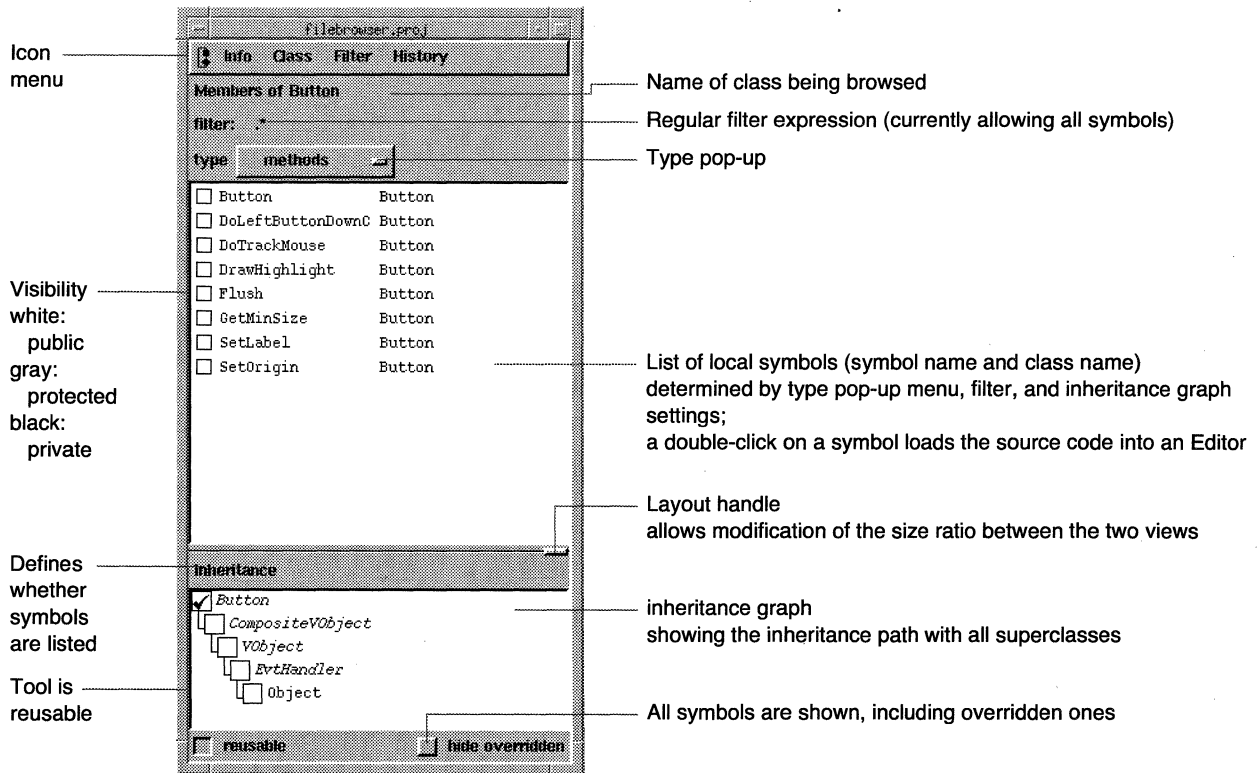
Project tree

The project tree shows the hierarchical project structure including subprojects. The only symbols shown are those whose project checkbox is checked. The checkboxes can be manipulated directly with the mouse or can be set with the Filter menu. A deep click (<Ctrl>click) on a project entry (not on the checkbox) will list symbols only from that project and will hide the symbols of all other projects.

CLASS BROWSER

The Class Browser shows a list of local symbols of the class currently being browsed. The content of this list is determined by the type selector, the settings of the inheritance graph view, and the filter field. The type selector is a pop-up menu that allows selection among methods, instance variables, local types, local enumerations, and friends. Class Browsers are invoked with the “Browse Class *class*” command from the Class menu.

The access privileges of methods and instance variables are indicated by the color of the squares located in front of them. Black means private, gray means protected, and white means public.



Inheritance Graph

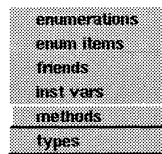
The inheritance graph view shows the graph consisting of the browsed class and its superclasses. The checkboxes in front of the classes show whether elements of those classes are listed. The checkboxes can be manipulated directly with the mouse or can be set with the Filter menu. A deep click (<Ctrl>click) on a class entry (not on the checkbox itself) will list elements of that class only and will hide the elements of all other classes.

Hide Overridden button

A further filtering mechanism is the possibility to hide overridden methods. This option can be set with the “Hide overridden” toggle button in the status line.

Type pop-up

The type pop-up specifies the type of the class elements shown in the list.



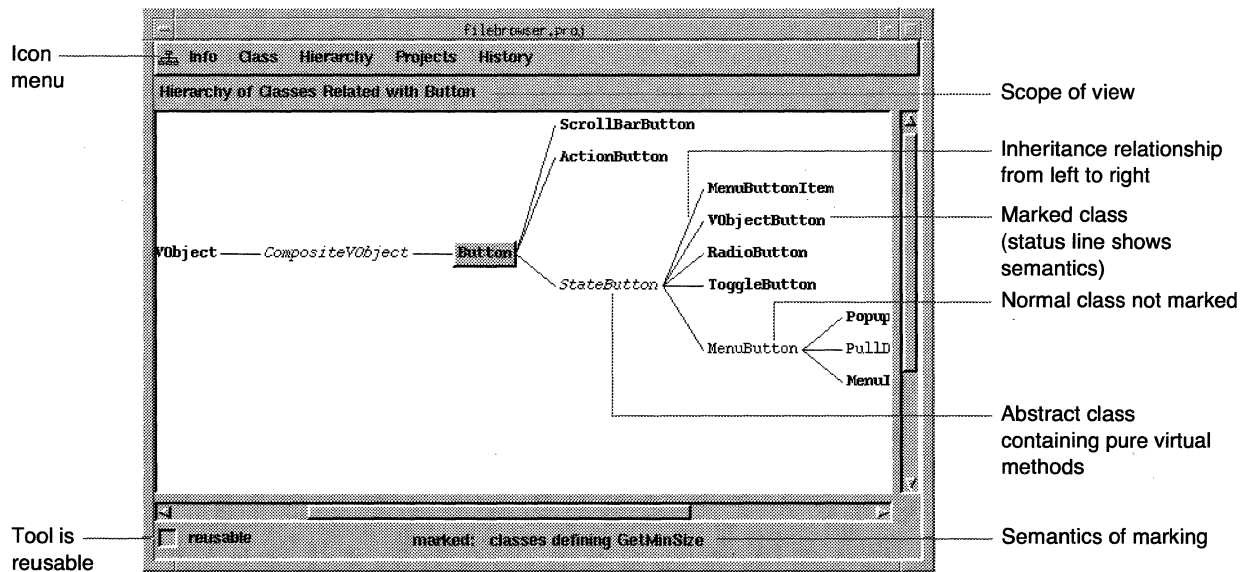
HIERARCHY BROWSER

The Hierarchy Browser is a graph browser designed to visualize a class graph. It either displays the entire class graph or only the superclasses and subclasses of the class indicated in the title of the browser view.

Hierarchy Browsers are invoked with the “Show Class in (Restricted) Hierarchy”, or the “Mark (Restricted) Classes Defining Method *method*” command in the Class menu. Another way is to issue “Hierarchy Browser” in the Icon menu.

There are two ways to mark a subset of the displayed classes. One way is to mark all classes that define a certain method. This kind of marking is obtained by issuing the “Mark (Restricted) Classes Defining Method *method*” command in the Class menu. The other way is to issue the “Mark Documented” command in the Info menu of the Hierarchy Browser.

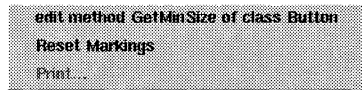
The semantics of the current marking are described in the status line.



Abstract classes (i.e. classes that define a pure virtual method) are displayed in italic font face.

Hierarchy menu

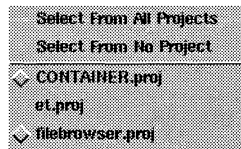
The Hierarchy menu serves to issue a set of commands related to the Hierarchy Browser.



Edit Method method of Class class	Loads the implementation of the corresponding method of the marked class into an Editor.
Reset markings	Resets the currently active markings.
Print...	Opens a Print dialog for printing.

Projects menu

The Projects menu makes it possible to show only classes of certain projects and to hide others. Hidden classes in the class graph are represented by a '+' sign.

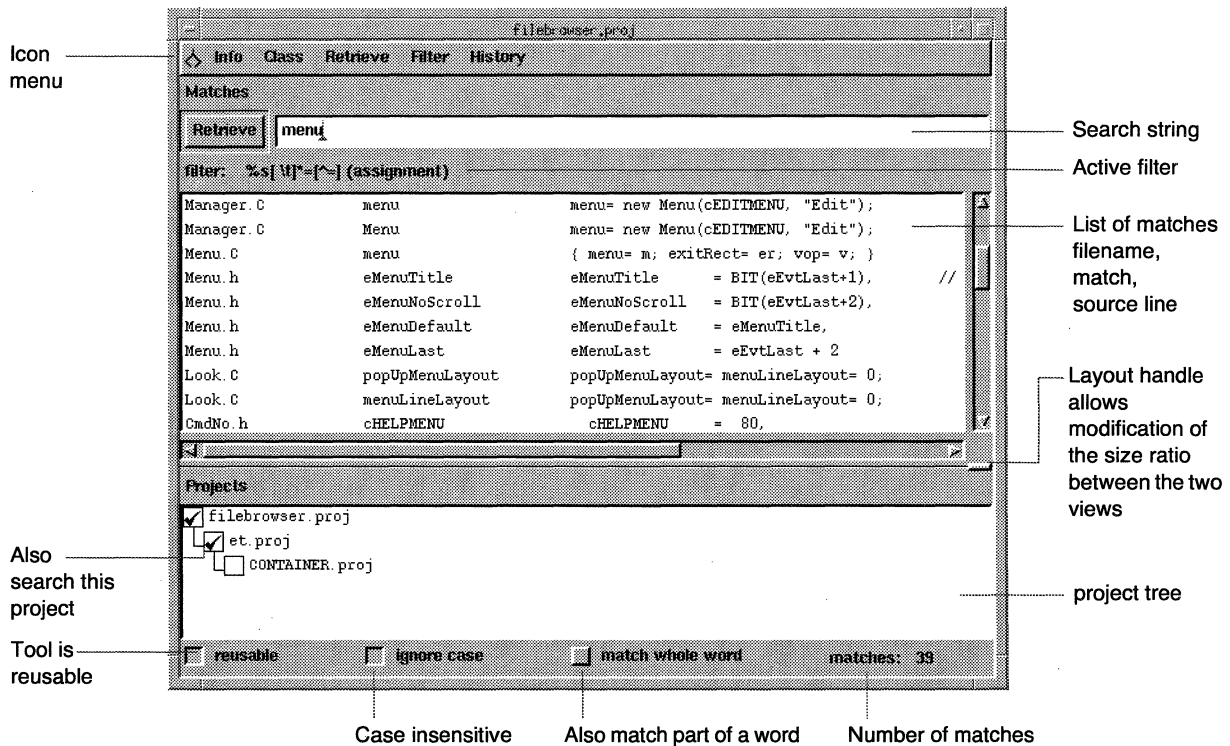


Select From All Projects	Shows the classes of all projects in the hierarchy view.
Select From No Project	Hides all classes except abstract classes.
Project entries with a toggle button	Hides or shows the classes of projects individually. All Project entries can be manipulated with the "Select From All Projects" and "Select From No Project" menu entries.


RETRIEVER

The Retriever shows a list of matches and a project tree view. The information about matches consists of the corresponding source file, the string that was matched, and the source line containing the match. The matches are obtained by a regular-expression-based search in all source files of the projects marked in the project tree view, and filtered with the active filter expression. In other words, the Retriever (like a super-grep in UNIX) starts a full text search over the project files and issues flexible semantic filtering as a second stage.

A Retriever is obtained by choosing the “Retrieve...” entries from the Info menu, or by issuing “Retriever” from the Icon menu.



The search process can be triggered either from the Retrieve menu by pressing the Retrieve button, or by pressing <Enter> after the regular expression defining a query. After the first retrieve, the source code is cached and all further queries are much faster. Caching can be switched off in the Preferences dialog (see “Preferences” on page 231). A Progress window indicates the progress of the search.

 **NOTE** The Retriever is the only SNIFF+ tool that is not updated after changes are applied to the source code.

Project tree

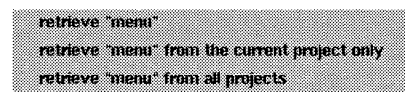
The project tree shows the hierarchical project structure including subprojects. Only symbols of projects are shown where the checkbox is checked. The checkboxes can be manipulated directly with the mouse or can be set with the Filter menu. A deep click (<Ctrl>click) on a project entry (not on the checkbox) will list symbols only from that project and will hide the symbols of all other projects.

Status line

Ignore Case	Specifies whether the search is case sensitive. The default is case sensitive search.
Match whole word	Specifies whether the search string must match a whole word. The default is that the search string is not restricted to being a whole word.
Matches	Displays the number of matches.

Retrieve menu

The Retrieve menu serves to trigger the retrieve process.



Retrieve selection	Triggers retrieving based on the current selections in the project tree view.
Retrieve selection From The Current Project Only	Triggers retrieving from the root project only.
Retrieve selection From All Projects	Triggers retrieving from all projects.

Filter menu

The Retriever's Filter menu consists of the standard Filter menu described in "Filter menu" on page 212 and a set of extendable semantic filters. These semantic filters are predefined regular expressions that serve to sensibly restrict the number of matches obtained from textual searches.

Set Filter...	>f
Reset Filter	>g
Select From "filebrowser.proj" Only	>o
Select From All Projects	>a
call	
assignment	
comparison	
new	

Predefined filters are:

Call	Lists only matches where the matched string is a method or procedure call.
Assignment	Lists only matches where the matched string is assigned a value.
Comparison	Lists only matches where the matched string is part of a comparison.
New	Lists only matches where the matched string is preceded by "new".

Additional filters can be added or the four standard filters can be overridden by providing a file consisting of a sequence of lines of string pairs delimited by double quotes (""). The first string is added to the menu and the second string is the filter which is inserted on selecting the corresponding menu entry. The Preferences dialog can be used to tell SNIFF+ where to find the filter extension file (see "Preferences" on page 231, "Files created and used by SNIFF+" on page 239, and "Appendix A. GNU Regular Expressions").

In formulating a filter criterion, the string "%s" can be inserted several times. It will be expanded with the actual search string.

EDITOR

SNIFF+ offers two possibilities for editing source code:

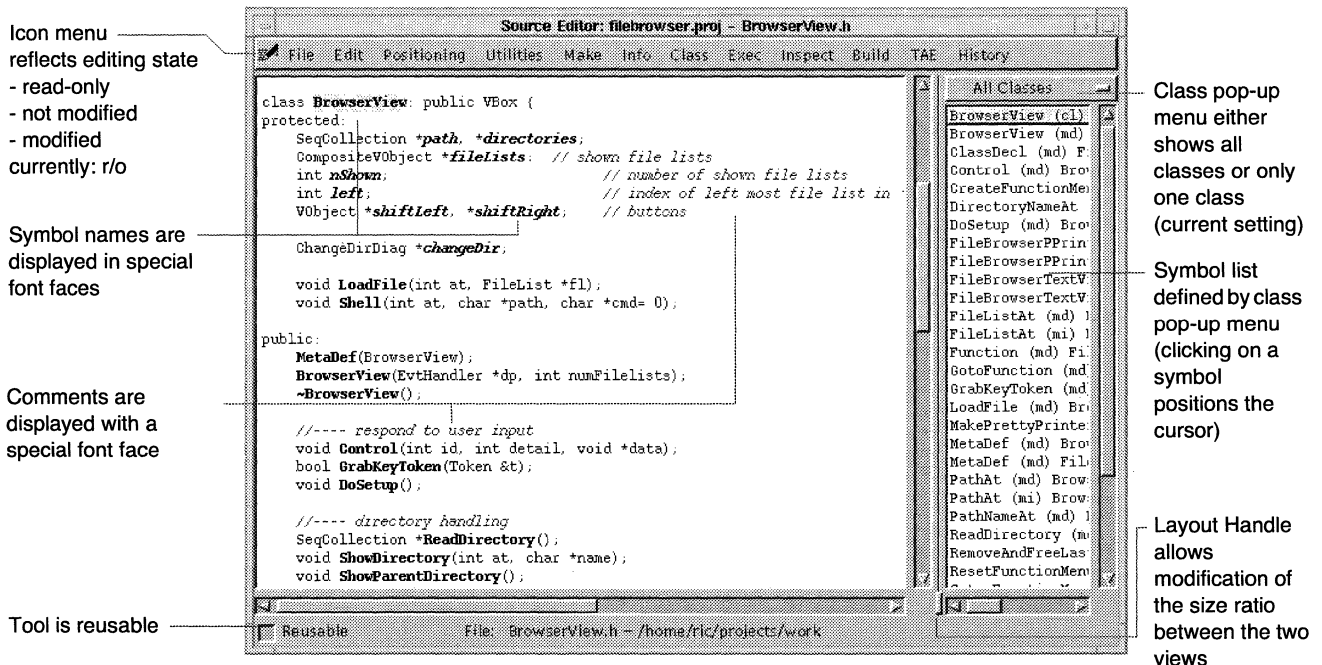
- ✦ SNIFF+'s own integrated Editor.
- ✦ An interface to standard Emacs version 19 or later (see “Emacs integration” on page 250).

This section describes how to work with the integrated Editor.

SNIFF+'s Editor consists of a WYSIWYG text Editor and a list of classes, methods, and functions defined in this file. This list speeds up the positioning by displaying the source code when a symbol is selected.

The Editor partially understands C/C++ syntax and can print comments and symbols with a different typeface. Which fonts and colors should be used for which symbols, the line spacing, and other attributes of the Editor can be defined in the ETRC file (see Appendix B, “ETRC File Entries”).

Besides the standard editing functionality, the Editor provides support for copying and moving the selection by direct manipulation, and it selects the text between matching characters such as brackets and quotes.



When a file is edited, the icon of the Icon menu changes to a warning sign until the file is saved.



File is not writable



File is writable



File is modified

The entered text is automatically reformatted. The time interval between reformatting can be set in the ETRC file (see Appendix B, “ETRC File Entries”).

Symbol List

The Symbol List is constrained by the Class pop-up and shows the list of:

- Method declarations “md” and implementations “mi”
- Class definitions “cl”
- Functions “f”

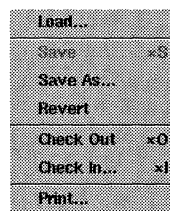
The Editor is positioned at the symbol by clicking on an entry in the Symbol List. A deep click (<Ctrl>click) on a declaration entry will position the Editor at the implementation and vice versa.

Class pop-up

The class pop-up scopes the Symbol List to either show only symbols of one class or to show all symbols of this file. This feature eases navigation when there is more than one class defined in a file.

File menu

The Editor's File menu contains standard commands for loading and saving files.




Load...	Opens a File dialog, which prompts for the name of the file to load.
Save	Saves the modified file (the option is only enabled when the file is modified). During the save the file is parsed and SNIFF+'s symbol table is updated. All tools are updated automatically to reflect the changes made to the file. A backup file can be created on every save (see “*.Document.MakeBackup(Bool):” in Appendix B, ETRC File Entries”).
Save As...	Opens a File dialog, which prompts for a new name of the file to save.
Revert	Reverts to the last saved version of this file (the option is only enabled if the file has been modified).
Check Out	Checks out and locks the latest version of this file. The protection of the file is set to writable. See also “Project Editor with locking information shown” on page 192.
Check In...	Checks in the currently loaded file. A Log message dialog is opened prompting for the log message for the newly saved version. The file is checked in as the newest version of the currently checked out branch. The file is saved before it is checked in. After the file is checked in, the protections of the working file are set to read only. (See also “Project Editor with locking information shown” on page 192.)
Print...	Opens a Print dialog for printing the file (see “Print Dialog” on page 175).

Edit menu

The Edit menu serves to issue standard commands for selecting, cutting, copying, and pasting text. Furthermore, it provides the (Un)Nest and (Un)Comment commands, which serve to shift the current selection tabwise or to put comment marks in front of the current selection.

Undo Move Text	<Z
Redo Paste Text	
Cut	<X
Copy	<C
Paste	<V
Select All	<A
Nest	<H
Unnest	<U
Comment	
Uncomment	

Undo command	Undoes the last change (<i>command</i>) to the text. The number of remembered undoable commands can be specified in the preferences file (see Appendix B, ETRC File Entries”).
Redo command	Redoes the last undone change (<i>command</i>).
Cut	Cuts out the current selection into the paste buffer (entry is only enabled if there is an active selection).
Copy	Copies the current selection into the paste buffer (entry is only enabled if there is an active selection).

 **NOTE** The Undo, Redo, Cut, Copy, and Paste commands are also accessible from the Edit pop-up menu, which appears in the text view when the right mouse button is pressed.

Paste	Pastes the paste buffer into the text at the current cursor location. If the cursor is a selection, the selection is replaced by the pasting. (The entry is only enabled if the paste buffer is not empty).
Select All	Selects the complete file contents.
Nest	Shifts the currently selected lines(s) one tab width to the right.
Unnest	Shifts the currently selected lines(s) one tab width to the left.
Comment	Inserts '//' comment at the beginning of the currently selected line(s).
Uncomment	Removes '//' comment at the beginning of the currently selected line(s).

Positioning menu

The Positioning menu provides commands for positioning in a text file, as well as the “Find/Change...” command.

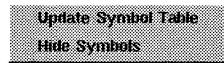
Edit Superclass Manager	
Edit Overridden Method Close	
Edit Implementation of Class	<E>
Edit Implementation File	<E>
Previous Position	<P>
Find/Change...	<F>
Find Again	<G>
Go To Line...	<L>
Next Match	<R>

If the cursor points at a symbol for which both declaration and implementation exist, <Alt>E switches between them. If not, the first entry is disabled and <Alt>E just switches between the declaration and implementation file.

Edit Superclass class	Jumps to the declaration of the superclass of the currently loaded <i>class</i> (this entry is only enabled if the cursor is positioned in the scope of a class that has a superclass).
Edit Overridden Method method	Loads into Editor the overridden <i>method</i> of the closest superclass that defines <i>method</i> .
Edit Declaration/Implementation of method	Toggles between the declaration and the corresponding implementation.
Edit Header/Implementation File	Toggles between the implementation file and the header file, and positions the cursor at the beginning of the file.
Previous Position	Jumps to the previous cursor position in this file.
Find/Change...	Opens a Find/Change dialog for finding or changing text. Regular expressions may be used (see “Find Dialog” on page 171).
Go To Line...	Opens the Go To dialog, which prompts for the line number.
Find Again	Jumps to the next match of the search string in the Find/Change dialog. This command also works if the Find/Change dialog is not open.
Next Match	Loads the next match of the most recently used Retriever.


Utilities menu

The Utilities menu serves to trigger various kinds of utilities.

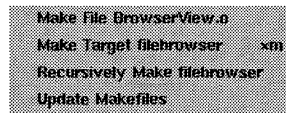


Update Symbol Table	Triggers an update of the symbol table after files of loaded projects were changed with tools other than SNIFF+. All files belonging to the project are checked and reloaded (reparsed) if they were modified and hence the symbol table is updated (see “Make menu” on page 189).
Hide/Show Symbols	Hides (or shows) the list of symbols used for fast positioning in the Editor.

Make menu

 **NOTE** Use the Build menu to execute Makeit in the Taligent Application Environment.

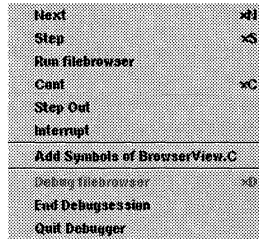
The make menu serves to issue three make commands. The command actually issued by these can be specified in the Preferences dialog and in the Project Attributes dialog (see “Project Editor” on page 187).



Make File objectfile	Obtains a Shell and starts “make <i>objectfile</i> ” in the project's source directory.
Make Target target	Obtains a Shell and starts “make <i>target</i> ” in the project's source directory.
Recursively Make target	Obtains a Shell and starts the “make” command for all subprojects bottom-up, depending on the attribute settings. Finally, “make <i>target</i> ” is called for the current root project.
Update Makefiles	Updates the dependency information for the makefiles of all editable projects. This command has to be issued only when a new include statement is inserted in one of the source files of the project. This command need not be issued when attributes or the project structure are changed, in which case SNIFF+ updates the makefile information automatically.

Exec menu

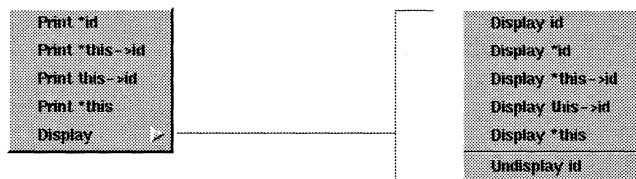
This menu does not currently apply to the Taligent environment. The Exec menu is a front end to the SNIFF+ Debugger commands and its entries are only enabled when the Debugger is started.



- Add Symbols of source file** Adds symbolic information for *source file*. This command searches for a cached object file in the generate directory of the project and recompiles the object with symbolic information before loading it.
- Debug Target target** Starts the debugger and loads the *target* executable. The entry is only enabled if the target name is set (see “Project Attributes Dialog” on page 199) and the target is executable.
- End Debugsession** Ends the current debugging session and quits the debugger backend (including the debugged application), but does not quit the SNIFF+ Debugger. The Debugger is iconified.
- Quit Debugger** Quits the SNIFF+ Debugger and the debugger backend (including the debugged application).

Inspect menu

The Inspect menu is a front end to the SNIFF+ Debugger commands and its entries are only enabled when the Debugger is started.



Build menu

The Build menu lets you build in the Taligent Application Environment.

Makeit Complete	Executes all standard phrases of a build for the current project directory and all its subdirectories.
Makeit (non-recursive)	Executes all standard phrases of a build for the current project directory
Makeit Testing Complete	Make the tests for the current project directory and all its subdirectories.
Makeit Includes	Ms the Includes phase for the current project directory and all its subdirectories.
Makeit Objects	Makes the Objects phase for the current project directory and all its subdirectories.
Makeit Libraries	Makes the Libraries phase for the current project directory ad all its subdirectories.
Makeit Binaries	Makes the Binaries phase for the current project directory and all its subdirectories.
Create FAST Makefile	Forces a rebuild of the .Make file for the current project directory and its subdirectories, using the -fast option.

See “Makeit” in Chapter 5 for more information.

TAE menu

The TAE menu gives you commands for starting, shutting down, and maintaining the Taligent Application Environment.


Start Taligent Application Environment	Starts the Taligent Application Environment. This takes awhile, so please be patient.
Shutdown Taligent Application Environment	Shuts down the Taligent Application Environment. You should be sure to execute this option before you try to use a reinstalled shared library.
Start Taligent Workspace	Starts the Workspace if it has been installed in your TaligentRoot.
MakeSOL	Runs the MakeSOL command. You should do this if you have added new shared libraries to the system. Be sure to shut down the system first!

See the Taligent *Installation and Release Notes* for more information on these features.

Custom menus

Custom menus allow the execution of customized commands in the Editor. You can have as many custom menus as you want. There are two ways to define a new menu:

- Modify the `EditorCustomMenu` config files located in `$SNIFF_DIR/config`.
- Add or modify `.EditorCustomMenu` config files in your home directory.

 **NOTE** If you specify the `.EditorCustomMenu` in your home directory, these files supersede the corresponding config file in `$SNIFF_DIR` (and therefore you lose access to any customization set by your site manager). A better strategy is to copy the required config file from `$SNIFF_DIR/config` to your home, rename it, and then modify it with new entries.


Placing entries in the Custom menu

Entries with no specified menus are placed in the Custom menu, as in:

```
shell "echo %s" "echo %s"  
shell "echo %F" "echo %F"  
shell "echo %l" "echo %l"  
filter "date" "date"
```

Adding multiple menus

You can add multiple menus by adding menu titles to the menu config file. To add a title, precede it with the “>” characters.

 **NOTE** This is a greater than symbol and a space.

This example specifies one menu : Misc.

```
shell "echo %s" "echo %s"  
shell "echo %d" "echo %d"  
shell "echo %f" "echo %f"  
filter "date" "date"  
> Misc  
shell "Command 1" "echo 1"  
shell "Command 2" "echo 2"
```

The first menu is called Custom, the second is called Misc.

In this next example, the first menu is called Echo, the second is called Misc.

```
> Echo  
shell "echo %s" "echo %s"  
shell "echo %d" "echo %d"  
shell "echo %f" "echo %f"  
shell "echo %D" "echo %D"  
shell "echo %F" "echo %F"  
shell "echo %l" "echo %l"  
filter "date" "date"  
> Misc  
shell "Command 1" "echo 1"  
shell "Command 2" "echo 2"
```

Debugging mode

This menu does not currently apply to the Taligent environment. After the command “Debug Target” is issued from the Exec menu, the Debugger is started and the Editor is in debugging mode. In Debugging mode the file is read only and a row of new buttons is added to the Editor window.



Run	Runs the debugged application from scratch.
Cont	Continues the interrupted execution.
Step	Single steps into the next function/method.
Next	Single steps over the next function/method.
Break In	Sets a break point at the current <i>selection</i> , whereby <i>selection</i> must be a valid function/method name.
Break At	Sets a breakpoint at the current cursor position (linewise).
Clear	Clears the breakpoint at the current line. The cursor must be positioned to a line with a breakpoint.
Print *	Prints the value pointed to by the current <i>selection</i> . <i>Selection</i> must evaluate to valid pointer.
Print	Prints the value of the current <i>selection</i> . <i>Selection</i> must evaluate to a valid variable.
this	Prints the value of the current object.
Stack	Opens a stack trace window and displays the current call stack.
Up	Goes one stack frame up in the call hierarchy. A reusable Editor is automatically positioned at the source location of the new stack frame.
Down	Goes one stack frame down in the call hierarchy. A reusable Editor is automatically positioned at the source location of the new stack frame.

Editing shortcuts and goodies

Selecting text

There are three ways to select text.

- ※ Characterwise by clicking and dragging with the mouse.
- ※ Wordwise by double-clicking and dragging with the mouse.
- ※ Linewise by triple-clicking and dragging with the mouse.

Marking of matching language items (brackets and quotes)

Double-clicking close to any of the following language elements:

single quotes- ' -
double quotes- " -
parentheses- (-
brackets- [-
braces- { -

causes the Editor to mark the code between this item and the matching one.

Fast copying

To avoid the overhead of copy/paste, a fast copy command can be used. Pressing the <Shift> and <Ctrl> keys at the same time and selecting a text to be inserted will copy this text to the current cursor position.

Another possibility is copying with direct manipulation.

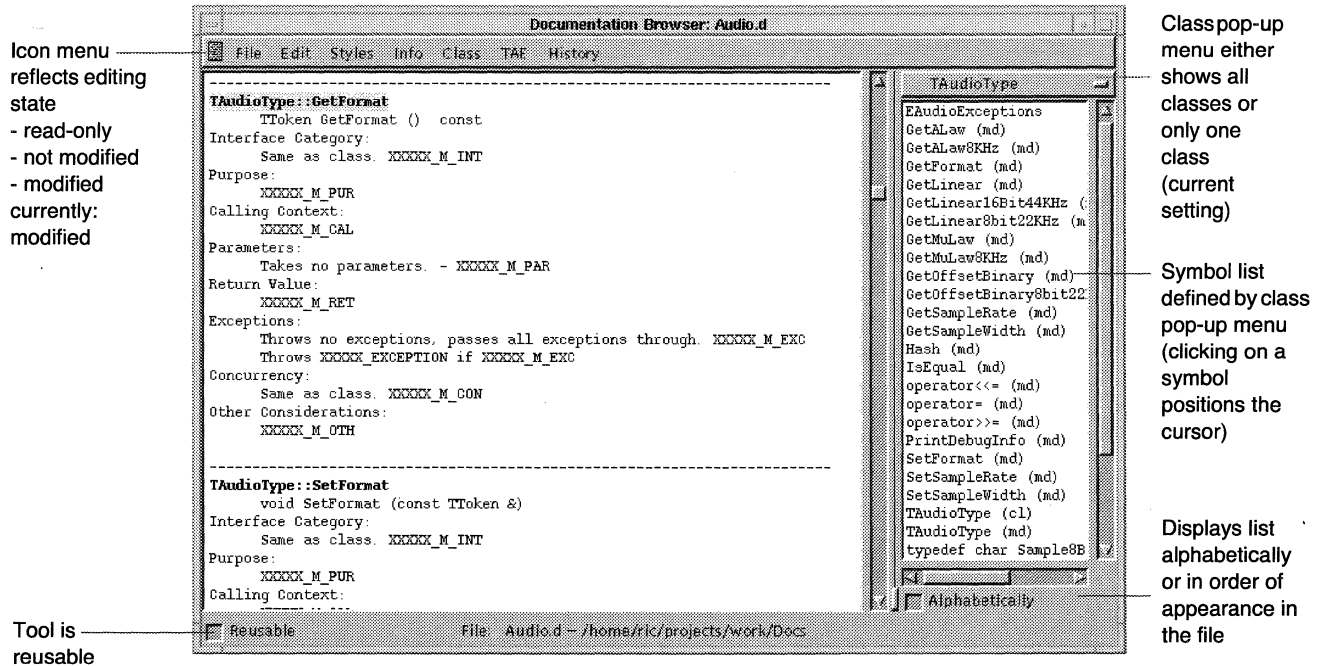
Copying and moving with direct manipulation

Clicking with the mouse on an active selection and dragging the text to a new position will move the selected text. Pressing the <Ctrl> key while dragging will copy the text instead of moving it.

DOCUMENTATION BROWSER

SNIFF+ lets you view and edit class and member descriptions in a special Documentation Browser.

Like the Editor, the Documentation Browser consists of a WYSIWYG text Editor and a list of classes, methods, functions, and data defined in this file. This list speeds up the positioning by displaying the description when a symbol is selected.



When a file is edited, the icon of the Icon menu changes to a warning sign until the file is saved.



File is not writable



File is writable



File is modified

The entered text is automatically reformatted. The time interval between reformatting can be set in the ETRC file (see "ETRC file entries" on page 261).

Symbol List

The Symbol List is constrained by the Class pop-up.

- ※ “md” indicates method declarations
- ※ “cl” indicates class definitions
- ※ “f” indicates functions

The Documentation Browser is positioned at the symbol by clicking on an entry in the Symbol List.

Class pop-up


The class pop-up scopes the Symbol List to either show only symbols of one class or to show all symbols of this file. This feature eases navigation when there is more than one class defined in a file.

File menu

The Documentation Browser's File menu contains standard commands for saving files.

Load	Opens a .d file directly, instead of using the Info menu. Displays a file dialog from which you can open the Docs directory and select the .d file you want to use.
Save	Saves the modified file (the option is only enabled when the file is modified). During the save the file is parsed and SNIFF+'s symbol table is updated. All tools are updated automatically to reflect the changes made to the file. A backup file can be created on every save (see Appendix C, “ETRC file entries”).
Revert	Reverts to the last saved version of this file (the option is only enabled if the file has been modified).
Check Out	(Do not use in this release)
Check In...	(Do not use in this release)
Print...	Opens a Print dialog for printing the file (see “Print Dialog” on page 175).

Edit menu	The Edit menu serves to issue standard commands for cutting, copying, and pasting text.
Undo command	Undoes the last change (<i>command</i>) to the text. The number of remembered undoable commands can be specified in the preferences file (see Appendix B, “ETRC File Entries”).
Redo command	Redoes the last undone change (<i>command</i>).
Cut	Cuts out the current selection into the paste buffer (entry is only enabled if there is an active selection).
Copy	Copies the current selection into the paste buffer (entry is only enabled if there is an active selection).
Paste	Pastes the paste buffer into the text at the current cursor location. If the cursor is a selection, the selection is replaced by the pasting. (The entry is only enabled if the paste buffer is not empty).

 **NOTE** The Undo, Redo, Cut, Copy, and Paste commands are also accessible from the Edit pop-up menu, which appears in the text view when the right mouse button is pressed.

Styles menu	The Styles menu formats text.
Default text	Changes the selected text to the default text font.
Emphasized text	Italicizes the selected text.
	See the <i>Class and Member Style Guide</i> for details on formatting class and member function descriptions.

Info menu See “Info menu” on page 179.

Class menu See “Class menu” on page 180.

TAE menu See “TAE menu” on page 220.

Custom menus See “Custom menus” on page 221.

SHELL

The Shell is a front end to the regular UNIX command line interface. It can be used for system-level manipulations, and it is used by SNIFF+ to issue make commands. Furthermore, it serves to select an error message and to trigger the marking of the corresponding source code with the “Find Error” command of the Shell menu.

Icon
menu

```

filebrowser.proj
Edit Info Class Shell Target
cd /home/chris/filebrowserII
sunsb3% make
etCC -g -I/Users/joe/Sniff2/et3/src -c BrowserView.C
BrowserView.C: In method 'BrowserView::BrowserView (class
ET_EvtHandler*, int)':
BrowserView.C:43: parse error before string constant
*** Error code 1
make: Fatal error: Command failed for target 'BrowserView.o'
sunsb3%
  
```

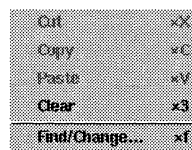
Make called by SNIFF+

Compilation error
click to it and select “Find
Error” from the Shell menu

Tool is
reusable

Edit menu

The Edit menu of the Shell contains the usual Cut/Copy/Paste/Find commands plus a Clear command.

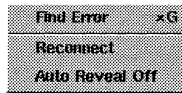


Clear

Clears the complete Shell buffer.


Shell menu

The Shell menu serves to issue three commands.

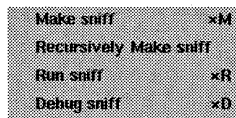


- | | |
|------------------------|--|
| Find Error | Filters the line containing the cursor. If it understands the error message format, it obtains an Editor and displays the corresponding source code. Section “Error formats file” on page 241 explains how to extend the list of understood error formats. |
| Reconnect | Reconnects to a new shell. |
| Auto Reveal On/
Off | Turns the auto-reveal feature on and off. If auto-reveal is on and input is typed or sent from a process, the Shell automatically scrolls to reveal the new text (this is the default). |

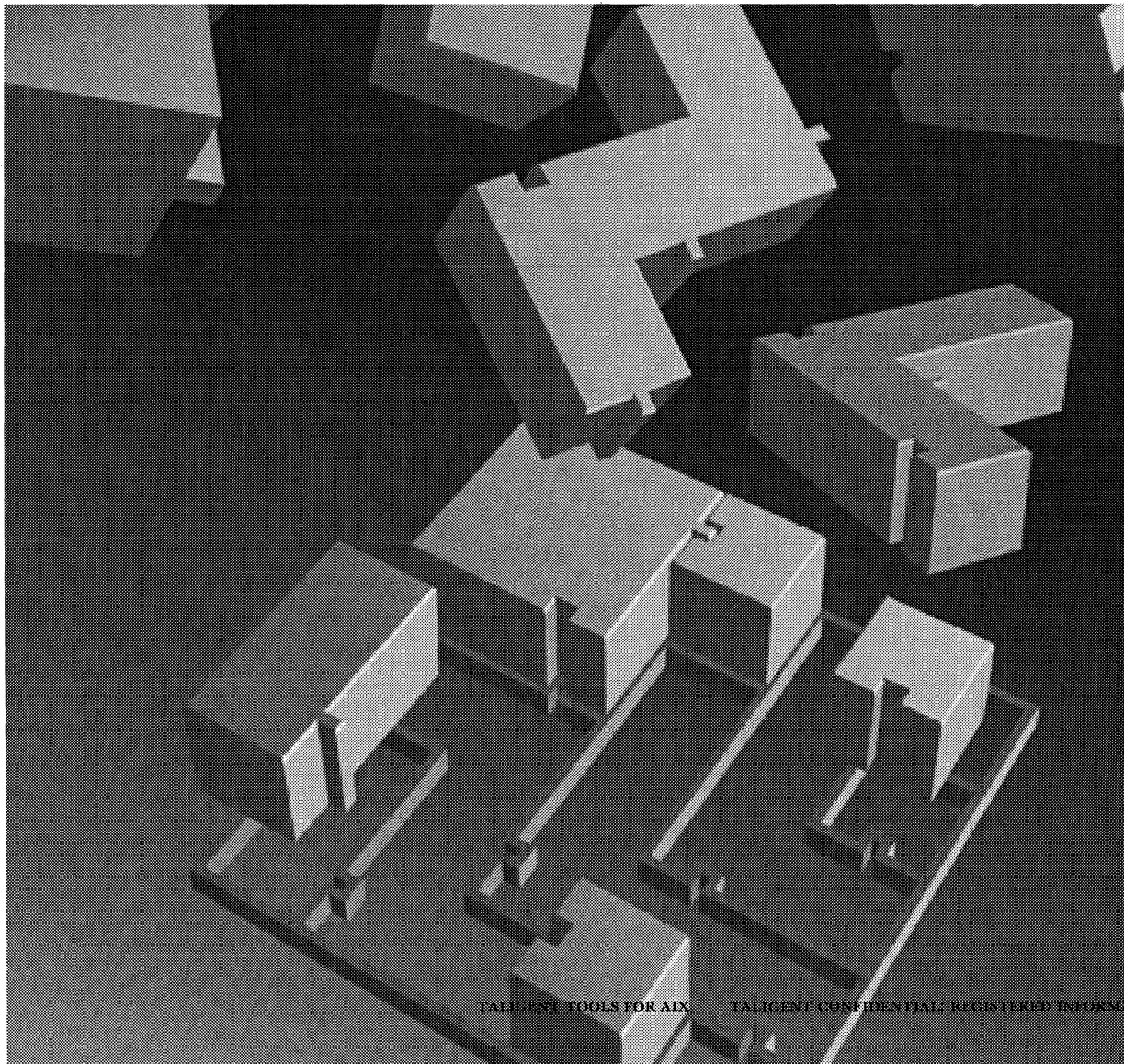
Target menu

 **NOTE** This menu does not apply to the Taligent Application Environment.

The target menu serves to make and run the target executable of the root project.



- | | |
|-------------------------|---|
| Make Target target | Obtains a Shell and starts “make <i>target</i> ” in the project's source directory. |
| Recursively make target | Obtains a Shell and starts the “make” command for all subprojects bottom-up, depending on the attribute settings. Finally, “make <i>target</i> ” is called for the current root project. |
| Run <i>target</i> | Obtains a Shell and executes <i>target</i> . |
| Debug Target target | Starts the debugger and loads the <i>target</i> executable. The entry is only enabled if the target name is set (see “Project Attributes Dialog” on page 199) and the target is executable. |

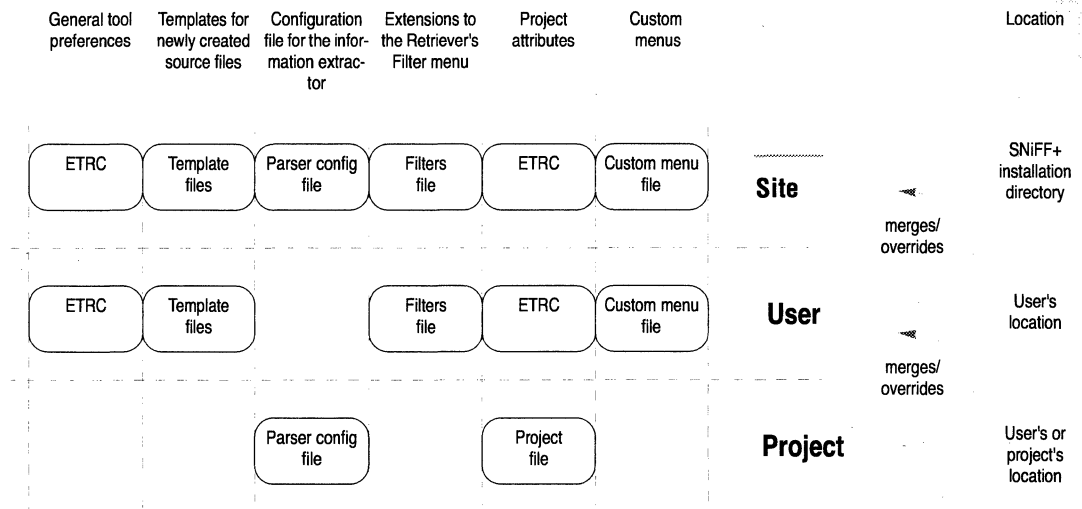


CHAPTER 14

CUSTOMIZING YOUR ENVIRONMENT

PREFERENCES

SNiFF+ supports the setting of preferences generally for a site (or project team), for each user individually, and for each project individually. Project preferences override user preferences, which in turn override site preferences.

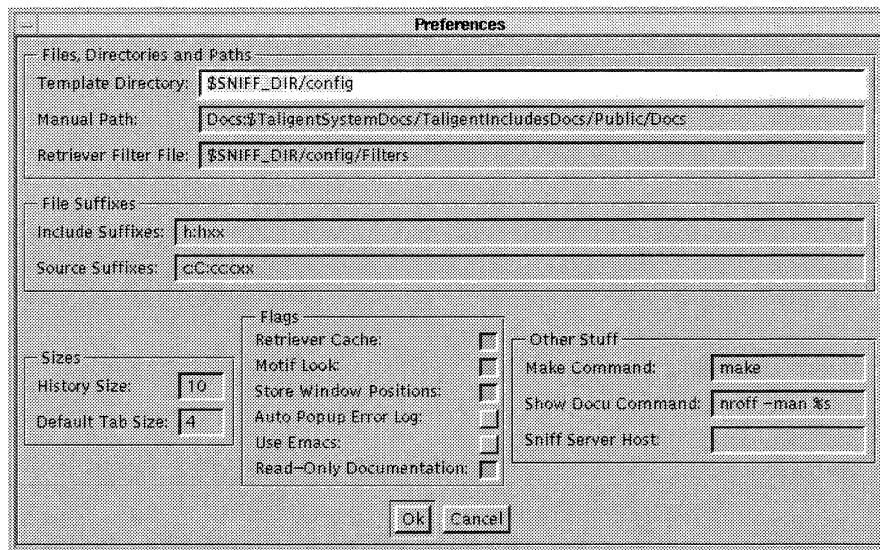


Each user of SNiFF+ has a private set of preferences. General preference settings are stored in a file called ETRC, located in the user's home directory. The most important and frequently changed settings of that file can be manipulated with the Preferences dialog. The other settings must be edited in the ETRC file directly (see "ETRC file entries" on page 261).

As shown in the illustration above, other preferences and configurations are stored in separate files.

Preferences dialog

SNiFF+'s Preferences dialog serves to browse and edit a number of settings that apply to all your projects. Some of these settings can be overridden in the Project Attributes dialog (see "Project Editor" on page 187). Generally, changes to the preferences take immediate effect. Exceptions to this rule are noted in the descriptions below.




SNiFF+ expands file and directory names using the C shell. Shortcuts such as the “~” or \$variables may therefore be used and are expanded correctly.

Files, Directories & Paths

Template Directory	Indicates the directory where the personal template files are stored. These template files are used when a new source file is created (see “Project menu” on page 190 and “Files created and used by SNiFF+” on page 239).
Manual Path	Contains a list of directory names separated by colons. These directories are searched for class and member descriptions when the “Show Documentation of” command of the Info menu is issued (see “Documentation Browser” on page 224).
Retriever Filter File	Indicates the file where extensions to the Retriever's Filter menu are stored (see “Files created and used by SNiFF+” on page 239 and “Retriever” on page 210).

File Suffixes

- Include Suffixes** Describes the valid suffixes of header files. Suffixes are separated with colons. The default is "h:hxx".
- Source Suffixes** Describes the valid suffixes of implementation files. Suffixes are separated with colons. The default is "c:cc:C:cxx".

 **NOTE** The point separating file name from suffix must be omitted.

Sizes

- History Size** Specifies how many previous states are kept in the history buffer of every tool.
- Default Tab Size** Defines the default size for tabulators. This attribute can be overridden for each project separately in the Project Attributes dialog.

Flags

- Retriever Cache** Specifies whether SNIFF+ caches files once they were searched by the Retriever tool. This option can speed up cross referencing considerably but it increases SNIFF+'s RAM requirement by the size of your files (which is frequently negligible).
- Motif Look** Specifies the look SNIFF+ selects at start-up. You can choose between Motif look and native ET++ look. Native ET++ look is superior on a black and white screen. Otherwise the selection of the look is a question of personal taste. A change takes effect on next SNIFF+ start-up.
- Store Window Positions** Defines whether the state of your current working environment is stored on closing a project and restored when it is loaded the next time. Default is to store window positions.
- Auto Popup Error Log** Specifies whether the Error log window (see "Error log window" on page 177) is automatically opened when a message is written to it. Default is not to open the window.
- Use Emacs** Determines the main Editor used by SNIFF+. You can choose between the SNIFF+ integrated Editor and Emacs.
- Read-Only Documentation** Changes documentation file from read-only to read-write. The icon on the Documentation Browser changes to indicate the file is writable. Toggle to change back.

Other Options

- Make Command** Specifies the command to be submitted to the shell when a make command is issued (see “Project menu” on page 186). The default is “make”. If you compile on a compile server, you can change the command, for example, to “on server make”, or you can provide your own shell script to do fancier things.
- Sniff Server Host** Indicates the host in the network where the information extractor process (sniffserver) runs. The default is no host (empty string), which means use a server process on the local machine.

TEAMWORK SUPPORT


SNiFF+ supports teamwork in several ways:

- Projects can be frozen. This prevents anybody from making changes to the project.
- Version control systems can be integrated (see “Version control” on page 254).
- Shared files can be overlaid.

Overlaying shared files

Files of a common subproject which are shared by several developers can be copied to the directory of a root project. If the overlay files attribute is enabled for the subproject and the superproject (see “Project Attributes Dialog” on page 199), the shared files are hidden by their copies. The developer works transparently with the copies, and on linking the target, the object files of the copies are linked. Hidden files are marked in the Project Editor with the string “hidden”, and during the loading of a project SNiFF+ notifies the developer in the Error log window about the files that are hidden.


Later on, the overlaid version can be merged into the shared version. This can be achieved by using the version control tools (see “Version control” on page 254) or by running the two versions through the UNIX diff facility and by manually merging the changes.

 **NOTE** Files are only overlaid if the Overlay Files attribute is set in the Project Attributes dialog of the project to be overlaid and the overlaying project.

INFORMATION EXTRACTOR (SNIFFSERVER)

When running, SNIFF+ consists of several processes: two of them are the SNIFF+ programming environment (`sniff`) and SNIFF+'s information extractor (`sniffserver`). The information extractor is a fuzzy C++ parser which analyzes C++, ANSI C, or Kernighan & Ritchie C source files and sends the programming environment a stream of information about the symbols defined and declared in the source code.

The `sniffserver` process can run on the local workstation or on any workstation on the network. The default behavior is to start the `sniffserver` process locally. If the process is to be started on another workstation, this can be indicated in the Preferences dialog (see “Preferences” on page 231).

 **NOTE** If SNIFF+ does not find a running `sniffserver` process, it tries to start one. If the server is to be started on the local workstation, the `sniffserver` executable is to be found in one of the command directories. If the server has to be started on a remote workstation, a shell script called `startsniffserver` has to be found in one of the command directories.

Running the sniffserver on a different host

Running the `sniffserver` process on a workstation other than the programming environment can make sense for several reasons.

If main storage is scarce on the local machine, it can make sense to put the load on a workstation with more RAM.

There are frequently fast server workstations on a network that can considerably shorten parsing time. This effect is not relevant during programming, when only single files are parsed at a time. But it can shorten start-up time when a large project has to be loaded.

Parsing on the workstation where the source code is physically stored reduces network traffic. Once again, this effect is not relevant during programming, when only single files are parsed at a time. It becomes relevant at start-up time, when a large project is loaded, or when many developers are working on projects located on the same server.

How to run the sniffserver on a remote host

Several things are necessary to run the sniffserver process on a remote host:

- ❖ The “Sniff Server Host” entry of the Preferences dialog (see “Preferences dialog” on page 232) must contain the name of the host where the sniffserver process is running.
- ❖ The file /etc/services must contain an entry similar to this:

```
sniffserver <port_num>/tcp
```

where <port_num> is an arbitrary unique tcp port number greater than 1024. If your computers run with Yellow Pages (YP) or NIS, then the entry should be made in the tables of the YP/NIS server. Please contact your system administrator to do that.

- ❖ If the sniffserver is not running on the remote machine, SNIFF+ executes startsniffserver, which has to be found in the command directories. startsniffserver is located in the <sniff_directory>/bin directory and is a shell script using the on command to start the sniffserver on the remote machine. Several restrictions apply for using the on command (see the UNIX manual pages). If the on command does not work, then you can start the sniffserver manually on the remote machine with the following command:

```
$SNIFF_DIR/bin/sniffserver <remote_hostname>
```

where <remote_hostname> is the host name of the machine running the sniffserver.

Dealing with preprocessor macros

SNIFF+'s information extractor 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.

SNIFF+ provides a mechanism to solve these kinds of problems by configuring the parser. For every project, you can write a file containing directives for the parser (see “Parser configuration file” on page 239).

The location of this file is defined with “Parser Config File” attribute in the Project Attributes dialog (see “Project Attributes Dialog” on page 199). After changing the configuration file, you should force a reparse of the project (“Project menu” on page 190).

The following examples illustrate how to configure the parser in case of problematic preprocessor macros.

Configuring the parser

Preprocessor macros to be ignored by the parser

The VIRTUAL macro is used in the NIH class library in class definitions like this:

```
class A : public VIRTUAL B
{ ... };
```

The VIRTUAL string can be ignored without losing information.

Strings to be ignored can be defined with `ignore string string` tuples in the parser configuration file. In the case of NIH this is:

```
ignore string VIRTUAL
```

Don't forget to reparse the project after the configuration file has been changed.

#ifdef directives to be resolved by the parser

Some class libraries use the preprocessor directive `#ifdef` to modify the code in a way that confuses the parser.

Examples are:

- ⌘ Different class definitions for the same class selected with an `#ifdef`:

```
#ifdef UNIX
    class someClass : unixBaseClass
#else
    class someClass : otherBaseClass
#endif
{ ... };
```

Since SNIFF+ normally parses the whole code without resolving `#ifdef`, it reads two class definition headers and just one actual definition.

To solve this problem 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. Alternatively, you could add this line to the configuration file:

```
undefine UNIX
```

The parser will ignore the line between `#ifdef` and `#else`.

- ⌘ 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 the whole code without resolving the `#ifdef`, it reads two opening and only one closing brace. To solve this problem add the following line to the parser configuration file:

```
define HUGE_INT
```

Another possibility to solve this particular problem is to remove the opening brace from the two for lines and put it after the `#endif` directive.

#if directives to be resolved by the parser

Sometimes it is necessary to resolve `#if` directives. For example:

```
#if defined (UNIX) || defined (VMS)
    class someClass : unixBaseClass
#else
    class someClass : otherBaseClass
#endif
{ ... };
```

The expression after the `#if` directive will be evaluated if it contains only 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, then the `#if` is not resolved (i.e. both branches will be parsed). Assuming that your configuration file contains

```
define AAA
undefine BBB
```

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
```

FILES CREATED AND USED BY SNIFF+

Project file

A project file describes a SNIFF+ project and is stored at a location indicated by the developer. A project file stores only structural information and attributes of a project. No source code or symbolic information is stored there. Project files are usually just a few KB in size (see also “SNIFF+ projects” on page 244).

ETRC file

The information manipulated in the Preferences dialog is stored in the ETRC file in the home directory of every SNIFF+ user (see “Preferences” on page 231 and “ETRC file entries” on page 261).

Parser configuration file

The parser configuration file contains special configuration instructions for the SNIFF+ information extractor (sniffserver). It can be defined for projects using preprocessor macros that semantically change the source code. For further explanations, see “Dealing with preprocessor macros” on page 236.

The file can contain lines with

- ※ Ignore string *string*

The parser just ignores *string* in the source code.

- ※ Define *symbol* or undefine *symbol*

The parser resolves `#ifdef` containing *symbol*. `#ifdefs` not containing *symbol* are parsed completely.

The location of the file can be specified with the Project Attributes dialog (see “Project Attributes Dialog” on page 199).

```
# Example ignore strings file
#
ignore string VIRTUAL      # for NIHCL
ignore string _C_ARG1     # for the License project
define UNIX                # resolve ifdefs for UNIX
```

Retriever filters file

The file describes filters that should be added to the Filter menu of the Retriever and consists of a sequence of lines of "" delimited string pairs. The first string is added to the menu and the second string is the regular filter expression that is applied on selecting the corresponding menu entry. In formulating a filter criterion, the string "%s" can be inserted several times. It will be expanded with the actual match for every retrieved source line.

The Preferences dialog can be used to tell SNIFF+ where to find the filter extension file (see "Preferences" on page 231, "Retriever" on page 210, and "ETRC file entries" on page 261).

```
# Example Retriever filters file
#
"call METHOD" "%s[().*;"
"declare class::METHOD" "[A-z0-9_ \t]+::%s[^;]+$"
"CLASS::method/var" "%s::.*;"
"class::METHOD/VAR" "[A-z0-9_ \t]::%s.*;"
"->METHOD" "->%s[()]"
"OBJECT->method" "%s->[A-z0-9_ \t]+[()]"
"->VAR" "->%s^[A-z0-9_]"
"OBJECT->var" "%s->[A-z0-9_ \t]+^[A-z0-9_]"
```

Template files

Template files are loaded into newly created project source files (see "Project Editor" on page 187). Templates must be called *template.extension*, whereby *extension* is one of the allowed extensions for header and implementation files. The location of template files and the list of allowed extensions can be specified with the Preferences dialog (see "Preferences" on page 231).

Custom menu file

SNIFF+ allows the definition of commands which are accessible from the Editor via the Custom menu. The file specifying the menu is called *.EditorCustomMenu* and is located in the user's home directory.

Site-specific custom menus may be defined in the config directory of the SNIFF+ installation. The name of the file must be: *EditorCustomMenu*. The files are loaded during start-up.

Syntax

CustomMenu={CustomMenuEntry}.

CustomMenuEntry=Descriptor MenuString Command
|Separator.

Descriptor="shell" | "debugger" | "filter".

MenuString=*string*.

Command=*string*.

Separator="-".

A “shell” command is executed in aSNIFF+ Shell.

A “debugger” command is sent to the Debugger.

A “filter” command is any kind of process. Its input is the current selection in the Editor and its output replaces the current selection.

A separator causes the insertion of a line in the menu. It is used for esthetic reasons only.

Commands are expanded as follows:

- %dProject file name
- %f Source file name
- %sCurrent selection
- %DSource directory
- %FBasename of source file
- %lLocking path without the RCS/SCCS extension
(used by the version control system to store the
version files)

Strings may be delimited with double quotes (“”) if they contain blanks.

Examples

```
# Example EditorCustomMenu file
#
shell "RCS diff" "rcsdiff -kk %l/RCS/%F,v %f"
shell "SCCS diff" "cd %D; sccs -d%l diffs %F"
-
filter Date date
shell "Load File Into vi" "cmdtool vi %f"
-
debugger "Info Files" "info files"
```

Error formats file

SNIFF+ integrates various compilers and other tools (like Purify). The Shell (“Shell menu” on page 228) and the Debugger (see “Icon menu” on page 177) are able to interpret the output messages of such tools based on a configurable error-formats file. The file \$SNIFF_DIR/config/ErrorFormats contains a list of regular expressions for the most common error formats. If the error messages of your compiler are not covered by an entry in that file, you can add the corresponding regular expression. Regular expressions are explained in “GNU Regular Expressions” on page 257.

Supplied ErrorFormats file

```
# SNIFF+ - regular expressions for compiler error messages
# "file.c", line 123
"\([^" ]+\)",[ ]+line[ ]+\([0-9]+\)
# file.c, line 123
\([^" ]+\),[ ]+line[ ]+\([0-9]+\)
# Purify: [line 123, file.c,
line[ ]+\([0-9]+\),[ ]+\([^" ]+\)
# file.c:123
\([^\: ]+\):[ ]*\([0-9]+\)
# file.c(123)
\([^" ]+\.\.[^\: ]+\)\([0-9]+\)
```

The parts of the regular expression that match the filename and the line number must be enclosed in a `\(\)` construct. Each regular expression must have exactly two such constructs.

Files generated by SNIFF+ and stored in the generate directory

For every project, SNIFF+ creates a directory that serves as a container for all project-dependent files generated by SNIFF+. Its location and how it can be changed is described in “Project Attributes Dialog” on page 199.

Make support files (dependencies.incl and ofiles.incl)

For every project a `dependencies.incl` file is generated that describes the include dependencies between the files of the project and its subprojects. An `ofiles.incl` file is generated that defines which object files have to be linked in building the current target. For a further description of how to use these files, see “Makefile Support” on page 249.

Symbol table files

SNIFF+ dumps symbol table files to the generate directory specified in the Preferences dialog (see “Preferences” on page 231). For a detailed description of symbol table persistency, see “Tuning and persistency of symbolic information” on page 243.

Window status files

Window status files are created by SNIFF+ for every user and are stored in the generate directory. These files are named `<project_name>.<user>.state` and store the position and contents of windows, the location of split handles and the cursor positions when the project is closed. The next time the project is opened the file is read in and the windows are restored.

TUNING AND PERSISTENCY OF SYMBOLIC INFORMATION

SNiFF+'s tools always work with the newest symbol information since they use the central symbol table (database) that is held in memory. The symbol table always is up to date and refers to the newest source code. This is possible because SNiFF+ directly uses the source code to extract the symbolic information. The information extractor is a very fast and lean parser. Once a project is loaded and parsed, all queries are executed in memory only – that is the reason why SNiFF+ scales linearly, and even huge software systems can be handled efficiently.

For smaller software systems up to 25 KLOC, the project loading time is no problem and information can be extracted on the fly.

For bigger projects (more than 25 KLOC), information extraction from the source code on every project load would be too time consuming, even with the very fast information extractor.

Therefore SNiFF+ allows the symbol table to be efficiently persistent between SNiFF+ sessions. Symbol table persistency is fully transparent to the user.

Depending on the kind and size of project, the user can choose between two different persistency models.

File-level symbol persistency (default)

After the first information extraction of a newly created project, SNiFF+ stores binary symbolic files for each source file in the generate directory for the project. The binaries are compact and efficient images of the symbol table held in memory. On each project load, SNiFF+ checks whether binary symbol files exist and loads them directly into memory instead of extracting the information from the source files. Information extraction from the source code is still possible if the source file is more recent than the binary symbol file. This can only happen when the source has been changed with a foreign tool or the date of the files has been otherwise modified.

Project-level symbol persistency

For library projects that are never changed, it makes sense to dump a single symbol table file for the whole project. This project symbol file is even more efficient and also much faster in loading. A project symbol file can be dumped (or actively removed) with the “Dump/Remove Symbol Table” command of the Project menu of the Project Editor (see “Make menu” on page 18g).

After the project symbol table has been dumped, SNiFF+ transparently manages this symbol file. If the library status of a project for which a project symbol table exists is changed to writable, the symbol table file is automatically removed and file-based symbol table persistency is used.

Comparison of project loading times

The following table relates the times needed to load a 60KLOC C++ project into SNIFF+ (the files of the project are located on the local system hard disk):

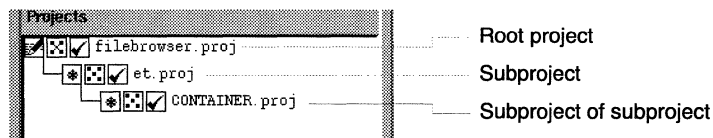
Kind of symbol loading	Relative loading time
Loading the symbols via the information extractor (parsing all source files)	100 %
Loading individual symbol files (file-level symbol persistency)	70 %
Loading one project symbol table dump (project-level symbol persistency)	40 %

The gain for project-level symbol persistency can be even bigger for projects located on NFS file systems.

SNIFF+ PROJECTS**Projects in SNIFF+**

A SNIFF+ project is a collection of source files and possibly a collection of subprojects belonging to the project. Projects are described and saved in project files.

Saving a project into a project file stores all information into that file. Opening a project file opens the complete project and loads all symbolic information and restores the window status.



If a project (project files) is opened in SNIFF+, it always forms the root project. Root projects are shown as the root in the project tree view of the different tools. Subprojects are shown as descendants. A subproject can also be opened on its own.

In the example above, `filebrowser.proj` is the root project, `et.proj` is the subproject, and `CONTAINER.proj` is the sub-subproject.

et.proj can also be opened on its own and is then the root project and CONTAINER.proj would be its subproject.

On every project open, the symbolic information of all subprojects is loaded.

The Project Editor (see “Project Editor” on page 187) serves to define the structure, the source files and the attributes of a project.

The rest of this chapter describes how to create projects that have a complicated structure or that have header and implementation files in separate directories.

Declaration and implementation files in separate directories

SNIFF+ requires a project per directory. Sometimes software systems have declaration (.h) and implementation (.C) files separated in different directories, but you would like SNIFF+ to treat such systems as if the files were all in one directory. SNIFF+ should therefore manage the different directories transparently.

To achieve that:

- 1 Create a SNIFF+ project for the directory where the declaration files (.h) are stored.
- 2 Create a SNIFF+ project for the directory where the implementation files (.C) are stored.
- 3 Load the project containing the implementation (.C) files as subproject of the declaration (.h) files project.
- 4 Close both projects.

When you reopen the declaration files project, you will also get the implementation files. This method of creating subprojects and loading them in a main project can also be used with more than two subprojects.

Example (InterViews)

The declaration files of InterViews are in the subdirectory `src/include/InterViews`, and the implementation files in the subdirectory `src/lib/InterViews`.

To create an InterViews project:

- 1 Create a SNIFF+ project for the `src/include/InterViews` directory. Name it `IV`.
- 2 Create a SNIFF+ project for the `src/lib/InterViews` directory. Name it `IV.impl`.
- 3 Using the Project Editor of `IV`, load `IV.impl` as a subproject of `IV`.
- 4 Close `IV` and `IV.impl` and reopen the `IV` project.

Projects with many subprojects

If you are working on a big software project, you probably have a hierarchy of directories and subdirectories, each containing the files of subprojects. Creating these subprojects one by one, as SNIFF+ requires, may be a time-consuming task.

Therefore, we supply a tool called `genproj` that walks a directory tree downwards and creates project files for every subdirectory.

`Genproj` is given the name of the directory that is the root of your software system and it generates project files in every subdirectory.

`Genproj` accepts the following parameters:

```
genproj <source_dir> [-e] [-f] [-p <proj_name>]
                    [-d <destination_dir>] [-s <sniff_directory>]
                    [-i <ignore_dir>]
```

`<source_dir>` is the only mandatory parameter. It is the name of the root directory of your software project. `Genproj` will walk this directory downwards and generate project files for every subdirectory.

`<proj_name>` is the name you want to give to the root project. If you don't specify a project name, `genproj` will use the base name of the source directory.

`<destination_dir>` is the name of the directory where you want to keep the generated project files. If you don't specify a destination directory, the project files will be generated in the corresponding subdirectories.

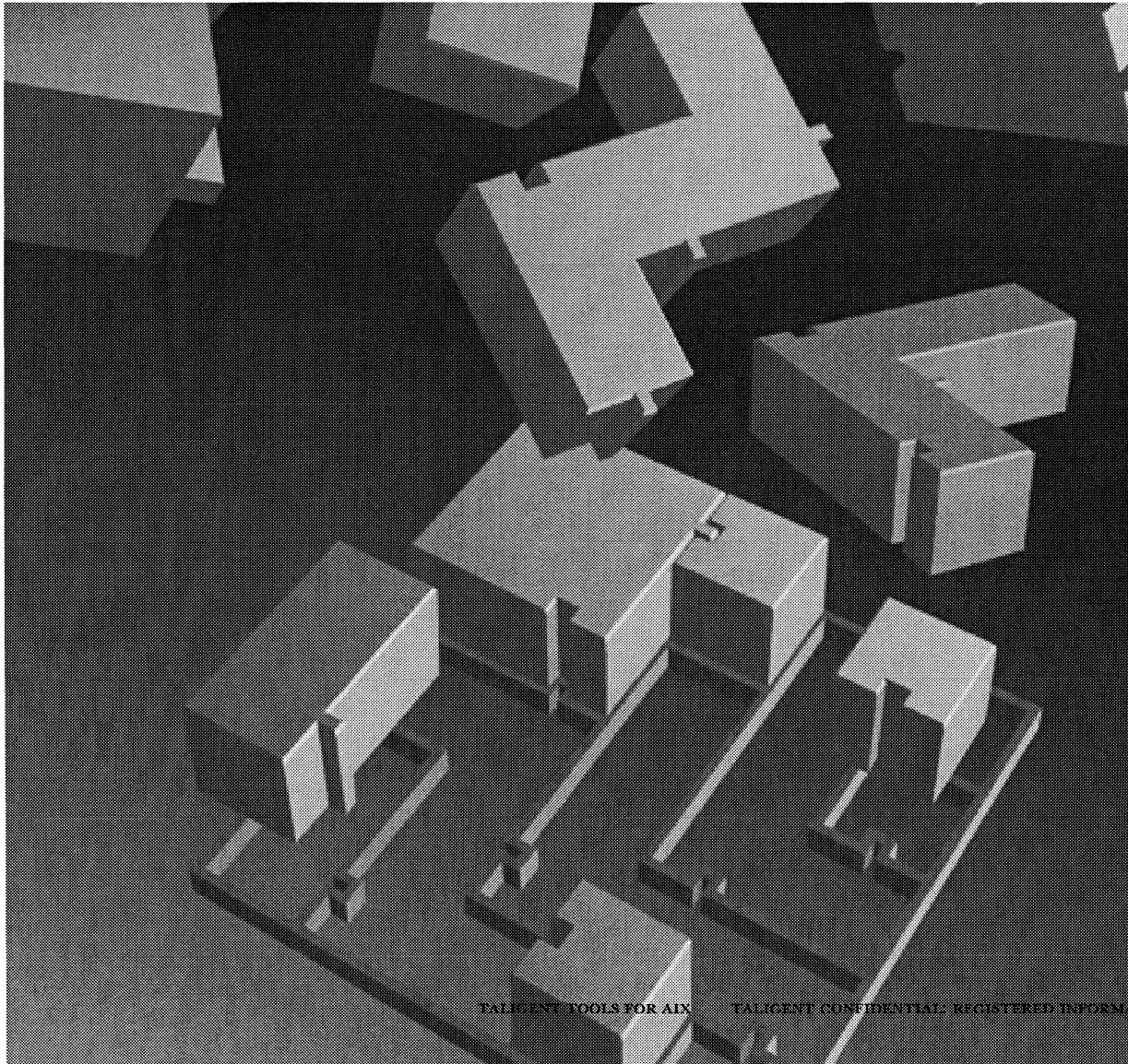
`<sniff_directory>` is the name of the directory where SNIFF+ stores the persistent symbolic information and other project-related data. The directory must already exist before starting SNIFF+. By default the directory is named `.sniffdir`, and it is created in the source directory of the project.

`<ignore_dir>` is the name of a directory (only the name of the directory and not the complete path) that should not be walked down. By default, `genproj` ignores all the directories whose name starts with a dot and the directories named `SCCS` and `RCS`. You can tell `genproj` to ignore additional directories by specifying the `-i` flag for each directory.

When specifying directories, you can use environment variables (but you must put their names in single quotes to prevent the shell from expanding them). The environment variable names will be copied literally into your project files. This will make the relocation of your software system easier, because you do not have to regenerate project files. Just modify the value of the environment variable and restart SNIFF+.

A project file will be generated only if the subdirectory contains source files (i.e. files whose suffixes are determined by “File Suffixes” attributes of the Preferences dialog; see “Preferences” on page 231), or if the subdirectory has other subdirectories that have source files. If you do want to have project files for subdirectories with no source files and no subprojects, then invoke genproj with the `-e` flag.

Genproj generates unique file names for projects. When it detects a name conflict, it uses the name of the parent project (with an underscore) as a prefix. The root project name is never changed. If you want genproj to prefix the names of all projects (except the root) with parent project names (even when there is no name conflict), then invoke it with the `-f` flag.



CHAPTER 15

SUPPORT FOR OTHER FUNCTIONS

MAKEFILE SUPPORT

SNiFF+ assumes that projects are compiled and linked with the `make` command or a similar facility. Make commands use makefiles, which describe the process of compiling and linking and the necessary options with a set of variables and rules.


SNiFF+ supports makefiles with two files (`dependencies.incl` and `ofiles.incl`) that make it possible to use the same makefile for different projects without modifying it.

The files are created in the generate directory of the project (usually `.sniffdir`). Whether the files are generated is determined by the Project attributes (see “Project Attributes Dialog” on page 199).

Dependencies (`dependencies.incl`)

Since SNiFF+ knows all about a project, it also knows about dependencies between the files of a project, even if the dependencies exist over project/subproject boundaries. On each save of the project file, SNiFF+ updates the dependencies and stores them in a file called “`dependencies.incl`” in the generate directory of the project (see “Preferences” on page 231).

This file stores the dependency information in a form understood by `make` and can therefore be included in the makefile.

 **NOTE** The “Update Makefile” command of the “Make” menu of the Editor triggers the update of the dependency information. You should issue this command when you add a new include file to one of the project sources.

```

# Example makefile showing how to include dependency and
# object file information generated by SNIFF+
#
.SUFFIXES: .C
CCFLAGS = -g
LDFLAGS =
CC      = etCC $(CCFLAGS)

.C.o: CC -dump $(CCFLAGS) -c $<
.c.o:
    cc $(CCFLAGS) -c $<

include .sniffdir/ofiles.incl

Object ----- myApp1: $(OFILES)
file list      $(CC) $(LDFLAGS) -o $@ $(OFILES)

include .sniffdir/dependencies.incl
Dependency --- # this line has to exist but it can be empty ----- $(OFILES) variable is
list                                                  set by ofile.incl

```

Object file list (ofiles.incl)

The second project-specific file created by SNIFF+ for inclusion in makefiles contains the list of object files for the target of the project. Like the `dependencies.incl`, on each save of the project file the `ofiles.incl` is updated and stored in the generate directory of the project. The `ofiles.incl` sets the make variable `$(OFILES)`, which can be used somewhere in the makefile, e.g., in the rule for linking the target (see makefile example above).

EMACS INTEGRATION

SNIFF+ offers two possibilities for editing source code:

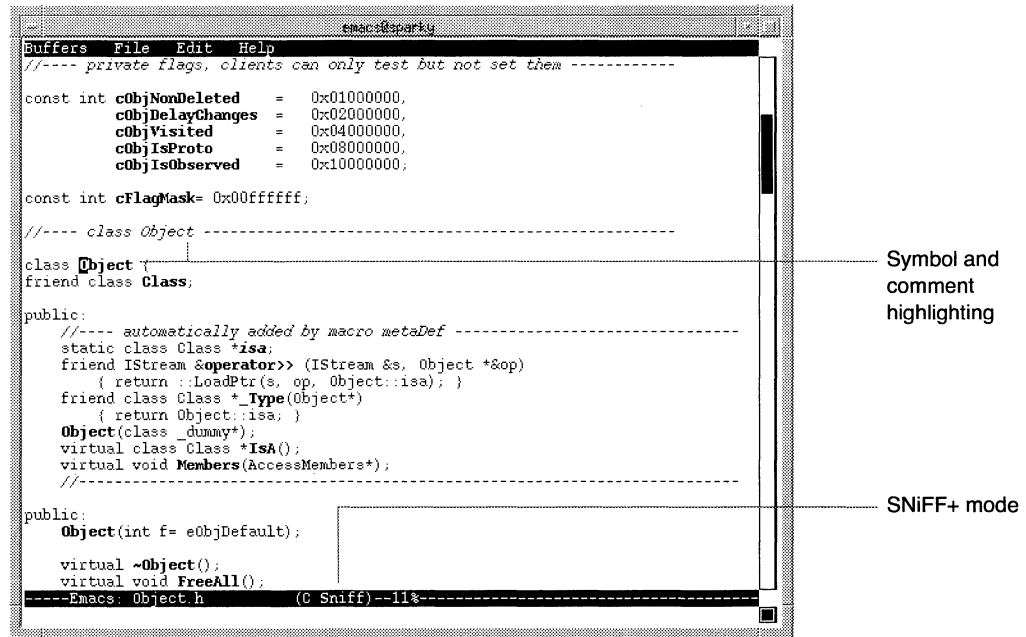
- SNIFF+'s own integrated Editor (see "Editor" on page 213).
- An interface to standard GNU Emacs (version 19 or newer).

This section describes how to integrate Emacs and how to work with it in the SNIFF+ environment.

The following features are available:

- Emacs is used for all SNIFF+ editing requests.
- SNIFF+ recognizes and updates all browsers when a file is saved in Emacs.
- SNIFF+ commands can be issued directly from Emacs.
- Emacs highlights symbols and comments like the integrated SNIFF+ Editor.

The following figure shows an Emacs running under X connected to SNIFF+:



Each user can have one Emacs to SNIFF+ connection active at a time.

Integrating Emacs

Integrating Emacs is fairly easy. All you need is:

- SNIFF+ up and running
- GNU Emacs (version 19 or later) installed at your site
- The sniff-mode.el file (part of the SNIFF+ package)

Telling SNIFF+ to use Emacs

The Preferences dialog of SNIFF+ contains a toggle button labeled "Use Emacs". Pressing the button tells SNIFF+ to use Emacs for all editing requests. If the button is pressed and no Emacs is actually connected to SNIFF+, a small dialog panel asks you whether to switch off the Emacs mode and use the own integrated Editor.

**Switching Emacs to
SNIFF+ mode**

The product package contains an Emacs-lisp file called `sniff-mode.el` that defines the SNIFF+ mode, how to talk to SNIFF+, and keyboard definitions for the available SNIFF+ commands.

The file is located in the directory `$SNIFF_DIR/config`.

To load the `sniff-mode.el` file at Emacs start-up, we suggest adding the following line to your `.emacs` file:

```
(load "$SNIFF_DIR/config/sniff-mode")
```

You can avoid the path specification by copying `sniff-mode.el` to the directory for site-wide Emacs-lisp files:

```
cp <sniff_directory>/config/sniff-mode.el /usr/local/lib/emacs/site-lisp
```

Whereby `<sniff_directory>` points to the root of the SNIFF+ installation. After you have done that, your `.emacs` file entry can look like this:

```
(load "sniff-mode")
```

**Connecting Emacs to a
running SNIFF+**

Whenever Emacs is started and is switched to the `sniff-mode`, a connection between SNIFF+ and Emacs has to be created. This is done by evaluating in the minibuffer:

```
M-x sniff-connect
```

**Disconnecting Emacs
from SNIFF+**

Emacs automatically recognizes when SNIFF+ is shut down and disconnects itself. A disconnection can be forced, though, by evaluating in the minibuffer:

```
M-x sniff-disconnect
```

**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. On the other hand, Emacs can send queries to SNIFF+.

**Positioning Emacs from
SNIFF+**

With Emacs as the main editor, you have almost the same possibilities to position quickly to a position in the source code as with the integrated Editor. Whenever you double-click in the browsers of SNIFF+ on a symbol or entry that has a relation to the source code, Emacs loads the corresponding source file and positions the cursor at the location.

When a new file is to be loaded, it is loaded into the currently active buffer (the buffer where the cursor is located). If the file is already loaded in a hidden buffer, Emacs is switched to that buffer.

Emacs highlights symbols in the source text by using different fonts. The symbolic information for this is supplied by SNIFF+. Highlighting can be switched off (see "Configuring the Emacs integration" on page 253).

SNiFF+ commands available in Emacs

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 the `sniff-mode.el` file (for changing the key bindings see “Configuring the Emacs integration” on page 253).

The following commands and bindings are available:

XXX table missing from p 108 XXX

If SNiFF+ cannot find an identifier to answer a query from Emacs, then a message is displayed in the echo area.

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 a file is loaded manually (via the `emacs load file` command), the buffer can be switched to SNiFF+ mode by evaluating the following command:

```
M-x sniff-mode
```

After the command is executed, all SNiFF+ key bindings are available and symbols are highlighted.

Configuring the Emacs integration**Changing key bindings**

The SNiFF+ key bindings are defined in the `sniff-mode.el` file. You can change the SNiFF+ key bindings as for any other Emacs key bindings.

Configuring symbol highlighting

EMACS is able to use different fonts. We use this feature to highlight symbols in source code. Emacs then is able to mimic the behavior of the integrated Editor.

The symbol highlighting is on by default, but can be switched off by setting

```
(setq sniff-want-fonts nil)
```

in your `.emacs` file (or interactively with `M-x set-variable`). Setting this variable to non-`nil` enables symbol highlighting.

The default font table for the highlighting is defined in the `sniff-mode.el` file. You can change the table by setting variables in your `.emacs` file. An example is:

```
(aset sniff-font-table 0 'bold-italic)
```

This will tell Emacs to use bold-italic face for comments. Please see the `sniff-mode.el` file for a full description of table entries.

If symbols and comments are not highlighted although the `sniff-want-fonts` variable is set, your Emacs might use a font that doesn't supply the necessary faces. To make Emacs using the courier font family, which should have all the different faces, try the following X resource (add the line to your `.Xdefaults` file):

```
emacs.font: -*-courier-medium-r-normal--*-120-75-75-**-**
```

How the Emacs integration works

To work together with the SNIFF+ environment, Emacs need not be changed. An Emacs configuration file is supplied with the SNIFF+ distribution. This file called `sniff-mode.el` contains Emacs-lisp code and tells Emacs how to communicate with SNIFF+.

Once the file is loaded, a new SNIFF+-mode is available in Emacs. Evaluating the function called `sniff-connect` builds up an interprocess communication and connects Emacs to the running SNIFF+ of the same user.

After SNIFF+ has been told to use Emacs as the main editor and after connecting, SNIFF+ uses Emacs to show and edit source code. Likewise, SNIFF+ provides commands to the Emacs user.

VERSION CONTROL

SNIFF+ supports various version control systems.

The following systems are supported:

- ✱ RCS version 5 or newer
- ✱ SCCS
- ✱ SNIFF+ internal locking. This is a simple file-based locking system without version control features

Each version system can be integrated into SNIFF+ with a flexible adapter architecture that provides a consistent user interface.

Version file and working file

A version file is located in the version control system repository and holds the complete history information of the file.

A working file is a checked out version and is the file that can be edited, saved, and checked in later.

**Restrictions in using
RCS and SCCS with
SNIFF+**

The following restrictions apply when using version control systems with SNIFF+:

- ✱ Every SNIFF+ project has one version control directory, but several SNIFF+ projects can share one version control directory.
- ✱ The directory names where the version files are stored can only have the following names:

Version control system	Directory
RCS	RCS
SCCS	SCCS

- ✱ For RCS the filename extension of the version file must be the default ,v.
- ✱ The RCS/SCCS commands must be available in the command search path of the sniff process.
- ✱ The user changing the version files must have write permission on the version control directories.
- ✱ Only strict locking is supported.
- ✱ Access list handling of RCS is not directly supported.
- ✱ Delete revisions is not directly supported.
- ✱ `rcsdiff`, `sccs diffs` and `rcsfreeze` commands are not directly supported.

Most of the above functionality can be made accessible in the custom menus of the SNIFF+ Editor (see “Custom menus” on page 221).

**Working with SNIFF+
version control and
locking**

SNIFF+ always extracts the symbolic information from the working files and not from the version files of the version control system. The main SNIFF+ tool to control and manage the version control is the Project Editor (see “Project Editor with locking information shown” on page 192). The check in and check out operations can also be accessed from the Editor (see “File menu” on page 225).

**How RCS and SCCS are
integrated**

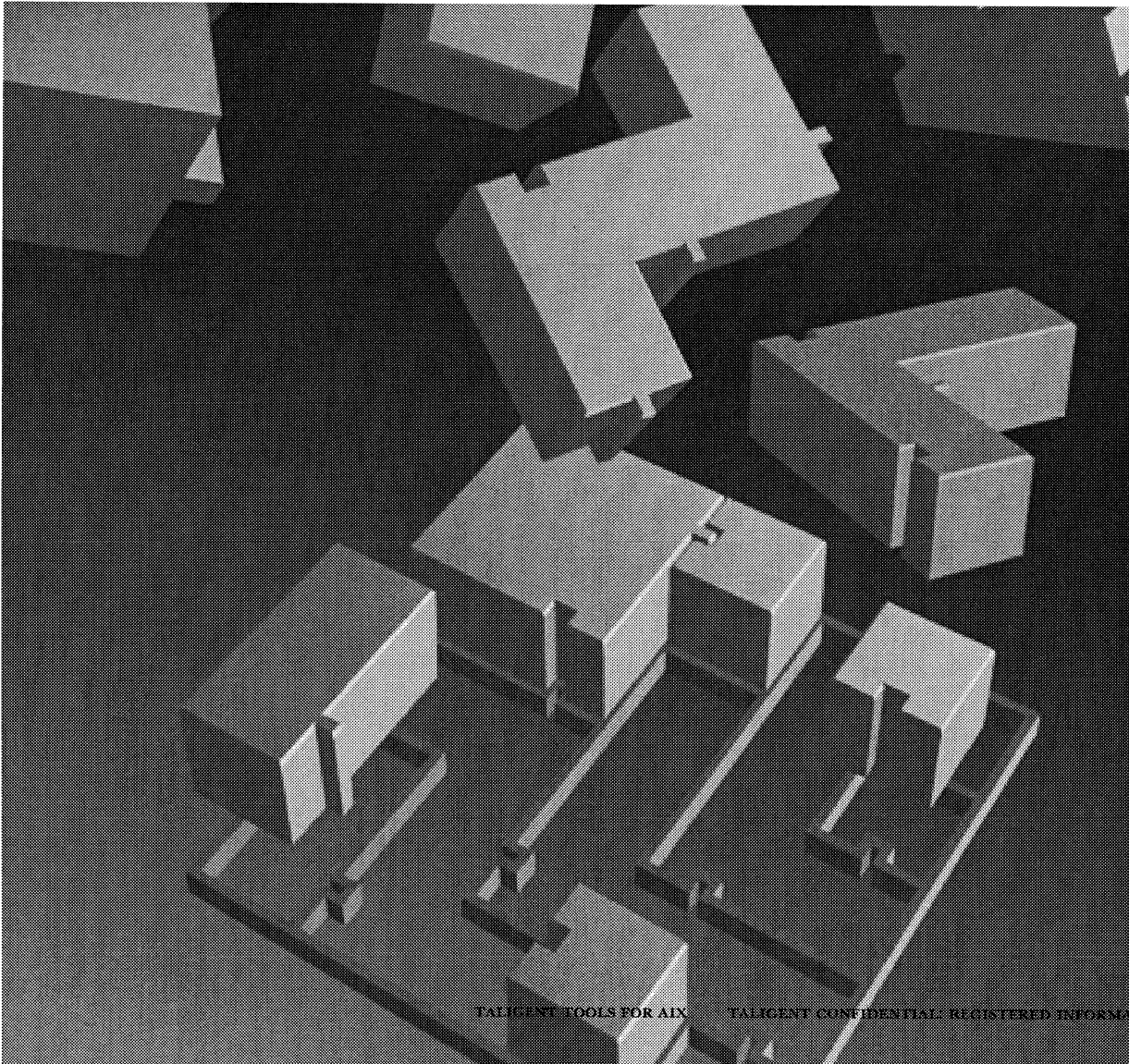
SNIFF+ provides a flexible adapter interface and consistent user interface to RCS and SCCS. All version control commands executed in SNIFF+ are translated to calls of the corresponding tools of the respective version control system.

SNIFF+ locking

The integrated SNIFF+ locking is a file-based locking system without version control features. It is intended to be used for projects where no version control is needed but where locking is important.

Information stored

The SNIFF+ locking stores the general description for a file and a log of changes to that file. The locking information is stored in a file called *sourcefile.lock* in the



APPENDIX B

GNU REGULAR EXPRESSIONS

GNU regular expressions are a very powerful means to specify patterns for filters and search strings in the various tools of SNiFF+. The syntax conforms to the regular expression syntax used in the EMACS editor.

Syntax

// Extended regular expression matching and search.

// Copyright (C) 1985 Richard M. Stallman

The GNU regular expression facilities are like those of most Unix editors, but more powerful:

- * -- * specifies a repetition of the preceding expression 0 or more times.
- + -- + is like *, but specifies repetition of the preceding expression 1 or more times.
- ? -- ? is like *, but matches at most one repetition of the preceding expression.
- \ -- \ specifies an alternative. Two regular expressions A and B with \ in between form an expression that matches anything that either A or B will match. Thus, "foo\bar" matches either "foo" or "bar", but no other string.
- \ applies to the largest possible surrounding expressions. Only a surrounding \(\dots\) grouping can limit the grouping power of \.
- Full backtracking capability exists when multiple \\'s are used.

- \ ... \--
- \(... \) are a grouping construct that serves three purposes:
- ※ To enclose a set of \ alternatives for other operations.
Thus, "\(foo\|bar\)x" matches either "foox" or "barx".
 - ※ To enclose a complicated expression for * to operate on.
Thus, "ba\(na\)*" matches "bananana", etc., with any number of na's (zero or more).
 - ※ To mark a matched substring for future reference.

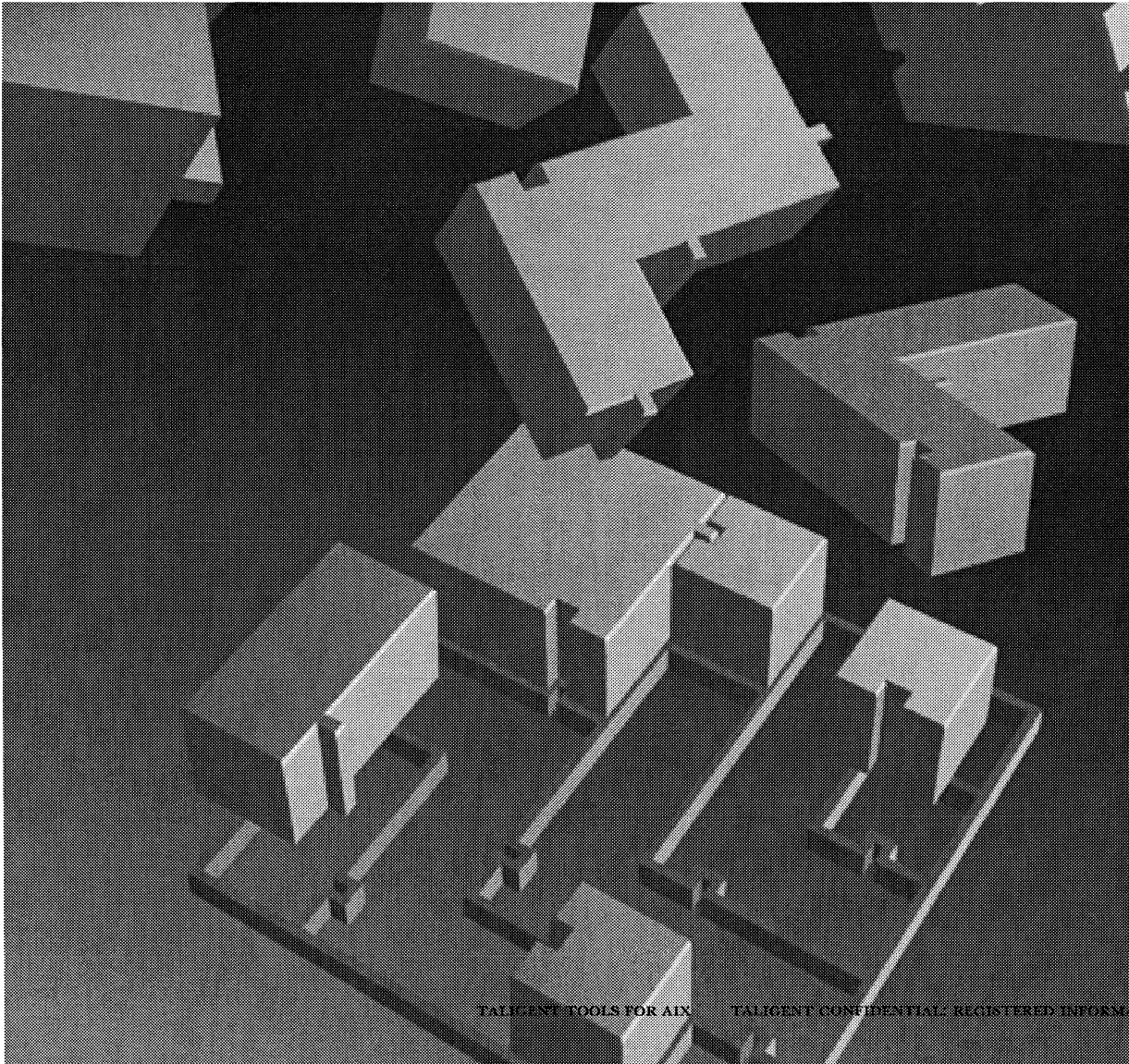
Application 3 is not a consequence of the idea of a parenthetical grouping; it is a separate feature that happens to be assigned as a second meaning of the same \(\ ... \) construct because there is no conflict in practice between the two meanings. The following is an explanation of this feature.

- \digit--
- After the end of a \(\ ... \) construct, the matcher remembers the beginning and end of the text matched by that construct. Then, later on in the regular expression, you can use \ followed by a digit to mean, "match the same text matched this time by the \(\ ... \) construct." The first nine \(\ ... \) constructs that appear in a regular expression are assigned numbers 1 through 9 in order of their beginnings. \1 through \9 can be used to refer to the text matched by the corresponding \(\ ... \) construct.

For example, "\(.*)\1" matches any string that is composed of two identical halves. The "\(.*)" matches the first half, which can be anything, but the \1 that follows must match the exact text.

- \ --
- Matches the empty string, but only if it is at the beginning of the buffer.
- \' --
- Matches the empty string, but only if it is at the end of the buffer.
- \b --
- Matches the empty string, but only if it is at the beginning or end of a word. Thus, "\bfoo\b" matches any occurrence of "foo" as a separate word. "\bball(s)\b" matches "ball" or "balls" as a separate word.
- \B --
- Matches the empty string, provided it is NOT at the beginning or end of a word.
- \< --
- Matches the empty string, provided it is at the beginning of a word.
- \> --
- Matches the empty string, provided it is at the end of a word.
- \w --
- Matches any word-constituent character. The editor syntax table determines which characters these are.

- **\W** -- Matches any character that is not a word-constituent.
- **\s<code>** -- Matches any character whose syntax is <code>. <code> is a letter that represents a syntax code: thus, "w" for word constituent, "-" for whitespace, "(" for open-parenthesis, etc. Thus, "\s(" matches any character with open-parenthesis syntax.
- **\S<code>** -- Matches any character whose syntax is not <code>.



APPENDIX C

ETRC FILE ENTRIES

The ETRC file stores all preference settings. Site-specific preferences are stored in the ETRC file located in the SNIFF+ installation directory. User-specific preferences are stored in the ETRC file located in the user's home directory.

Some entries of the user-specific ETRC file can be edited with the Preferences dialog (see "Preferences" on page 77). All other user-specific and all site-specific entries must be edited in the corresponding files directly.

DESCRIPTION OF ENTRIES

ETRC entries for all SNIFF+ applications (sniff and sniffgdb)

Resource name	Default value	Description
*.WindowSystem.Motif(Bool):	YES	Look Motif or ET++
*.WindowSystem.DoubleBuffer(Bool):	YES	Double buffering gives flicker-free screen updates
*.WindowSystem.ForceMonochrome(Bool):	NO	Display only 1 bit per pixel; force monochrome output
*.WindowSystem.MaxDepth(Num):	32	Maximum bits/pixel on color systems
*.WindowSystem.HighlightColor(RGBColor):	2 0 255 255 0 0 #yellow	Color for selections and highlights. Format: Don't-change Don't-change Red Green Blue Alpha. Alpha should always be 0. Range for RGB values: 0 - 255.
*.WindowSystem.WindowBackgroundColor(RGB Color):	2 0 190 190 190 0 #grey	Background color of windows. For format see WindowSystem.HighlightColor.

Resource name	Default value	Description
*.WindowSystem.ViewBackgroundColor(RGBColor):	2 0 255 255 255 0 #white	Background color of all views (also Editor text view). For contrast reasons you might also want to change Sniff.Code.Color etc. For format, see WindowSystem.HighlightColor
*.WindowSystem.DisableColor(RGBColor):	2 0 144 144 144 0 #dark grey	Color of disabled items. For format, see WindowSystem.HighlightColor
*.IAC.Debug(Bool):	FALSE	Interapplication communicator prints control messages to stderr
*.SysFont:	Chicago-Medium-12	System font: Family-Face-Size (possible values are listed in the ETRC file)
*.ApplFont:	Helvetica-Medium-12	Application font: Family-Face-Size
*.FixedFont:	Courier-Medium-12	Fixed font: Family-Face-Size for fixed text views (Editor, Shell, Debugger)
*.LineSpacing:	1	General line spacing for all fonts and all tools
*.TextView.CaretColor(RGBColor):	6 0 255 0 0 0 #red	Color of caret (cursor) in textview. For format, see WindowSystem.HighlightColor.
*.ShellText.UseStyles(Bool):	YES	Allow different styles in a Shell
*.CodeText.UseStyles(Bool):	YES	Allow different styles in source text view
*.CodeText.AllowGraphics(Bool):	YES	Allow graphical objects
*.CodeText.TabPos(Num):	4	Tabulator width for non-SNIFF+ files
*.CodeText.AutoIndent(Bool):	YES	Automatically indent lines
*.CodeText.WordWrap(Bool):	NO	Wrap words around lines if text gets too long
*.ScrollBar.Thickness:	16	Width of scrollbars

Resource name	Default value	Description
*.Document.MakeBackup(Bool):	NO	Create a backup copy (<i>filename%</i>) on a file save
*.Document.UndoLevel(Num):	99999	Number of undo levels
Doc.ItemName.Font	Helvetica-Bold-14	Font for item names
#Doc.ItemName.Color	2 0 190 190 190 0	Color of item names
Doc.ItemName.Alignment	2 # centered	Center alignment of item names
Doc.FieldName.Font	Times-Bold	Font for field names
#Doc.FieldName.Color	2 0 190 190 190 0	Color of field names
Doc.NormalParagraph.Font	Times	Font for normal text
Doc.NormalParagraph.Indentation	10	Indent for indented text
#Doc.Emphasis	3 # bold italic	Font for emphasized text
Doc.ObsoleteItem.Color	2 0 100 100 100 0	Color of obsolete items
*.SHELL:	/bin/csh	Shell used for the Shell

ETRC entries for sniff

Resource name	Default value	Description
Sniff.StoreState:	YES	Store the window positions and sizes between sessions
Sniff.MainWindow:	10:10:250:200	Location and size of the Workspace Manager window
Sniff.TabSize:	4	Default Tabulator width (can be overridden for each project)
Sniff.MakeCommand:	make	Command called on makes
Sniff.IncludePostfixes:	h:hxx	Recognized suffixes of header files (Format: ':'-separated list)
Sniff.SourcePostfixes:	c:cc:C:cxx	Recognized suffixes of implementation files (Format: ':'-separated list)
Sniff.Emacs:	NO	Use Emacs as editor

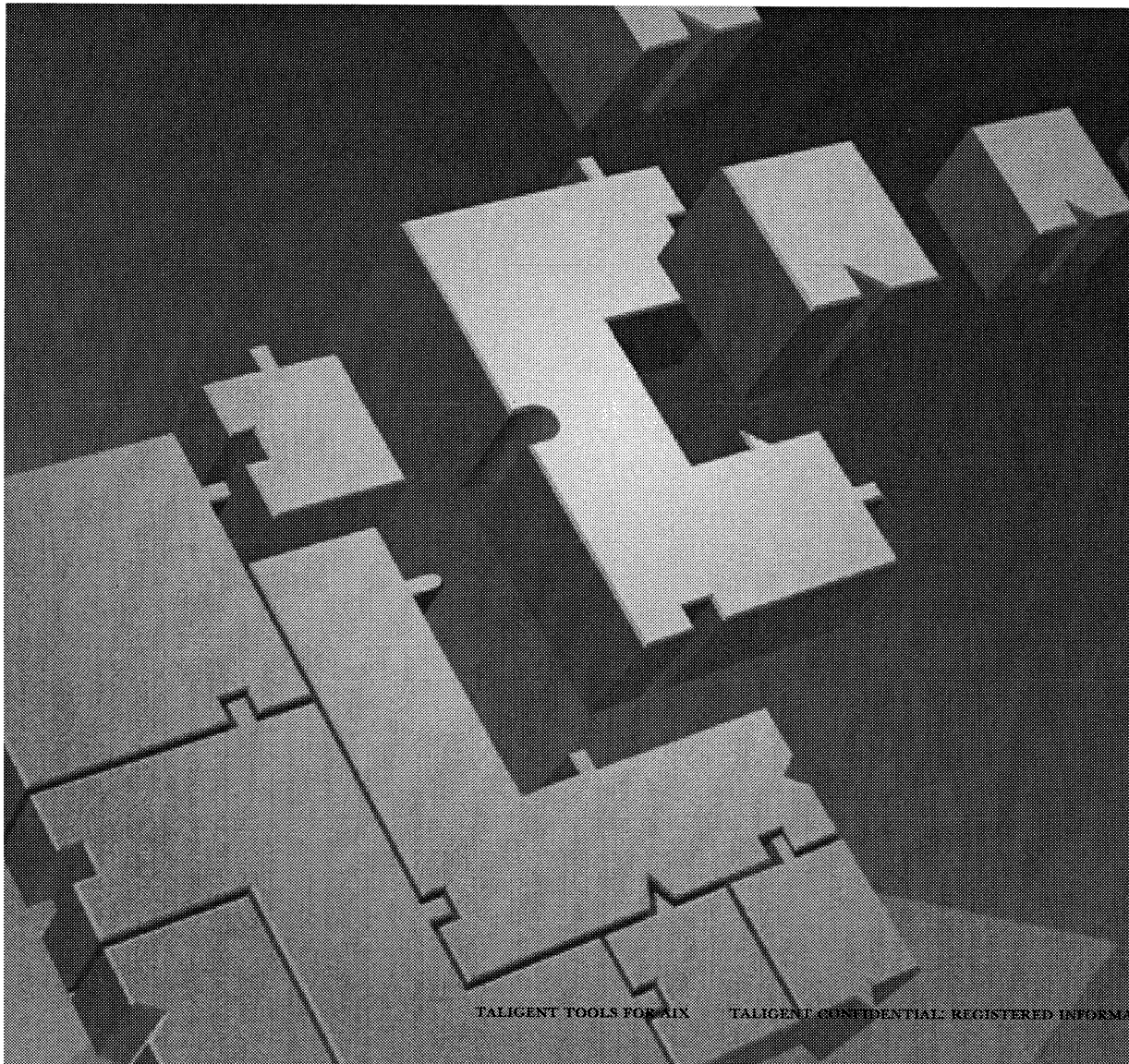
Resource name	Default value	Description
Sniff.ServerHost:		Host where the sniffserver is run
Sniff.ManualPath:		Search paths for documentation files (Format: ':'-separated list)
Sniff.FilterFile:		File defining additional filters for the Retriever
Sniff.RetrieverCache:	TRUE	Cache source files after the first search in the Retriever
Sniff.HistorySize:	10	Maximum number of recallable history entries in the History menus
Sniff.TemplateDir:		Directory where templates for newly created files are stored
Sniff.PopupErrorLog:	FALSE	The Error Log window is automatically opened when a message is written to
Sniff.Code.Font:	Courier-Medium-12	Editor code font: Family-Face-Size
Sniff.Code.Color:	2 0 0 0 0 0 #black	Color for normal code. For format see WindowSystem.HighlightColor
Sniff.Comment.Font:	Courier-Medium-12	Comment font: Family-Face-Size
Sniff.Comment.Color:	2 0 92 92 92 0 #dark grey	Color for comments. For format see WindowSystem.HighlightColor
Sniff.Macro.Font:	Courier-Bold-12	Macro font: Family-Face-Size
Sniff.Macro.Color:	2 0 92 92 92 0 #dark grey	Color for macros. For format see WindowSystem.HighlightColor
Sniff.Class.Font:	Courier-Bold-12	Class font: Family-Face-Size
Sniff.Class.Color:	2 0 92 92 92 0 #dark grey	Color for classes. For format see WindowSystem.HighlightColor
Sniff.InstVar.Font:	Courier-BoldOblique-12	Instance variable font: Family-Face-Size

Resource name	Default value	Description
Sniff.InstVar.Color:	2 0 92 92 92 0 #dark grey	Color for instance vars. For format see WindowSystem.HighlightColor
Sniff.MethodDef.Font:	Courier-Bold-12	Method definition font: Family-Face-Size
Sniff.MethodDef.Color:	2 0 92 92 92 0 #dark grey	Color for methods defs. For format see WindowSystem.HighlightColor
Sniff.MethodImpl.Font:	Courier-Bold-12	Method implementation font: Family-Face-Size
Sniff.MethodImpl.Color:	2 0 92 92 92 0 #dark grey	Color for method impls. For format see WindowSystem.HighlightColor
Sniff.Function.Font:	Courier-Bold-12	Function font: Family-Face-Size
Sniff.Function.Color:	2 0 92 92 92 0 #dark grey	Color for functions. For format see WindowSystem.HighlightColor
Sniff.Friend.Font:	Courier-Bold-12	Friend font: Family-Face-Size
Sniff.Friend.Color:	2 0 92 92 92 0 #dark grey	Color for friends. For format see WindowSystem.HighlightColor
Sniff.TypeDef.Font:	Courier-Bold-12	Type definition font Family-Face-Size
Sniff.TypeDef.Color:	2 0 92 92 92 0 #dark grey	Color for typedefs. For format see WindowSystem.HighlightColor
Sniff.Variable.Font:	Courier-Bold-12	Variable font: Family-Face-Size
Sniff.Variable.Color:	2 0 92 92 92 0 #dark grey	Color for variables. For format see WindowSystem.HighlightColor
Sniff.Const.Font:	Courier-Bold-12	Constant font: Family-Face-Size
Sniff.Const.Color:	2 0 92 92 92 0 #dark grey	Color for constants. For format see WindowSystem.HighlightColor

Resource name	Default value	Description
Sniff.Enum.Font:	Courier-Bold-12	Enumeration font: Family-Face-Size
Sniff.Enum.Color:	2 0 92 92 92 0 #dark grey	Color for enumerations. For format see WindowSystem.HighlightColor
Sniff.EnumItem.Font:	Courier-Bold-12	Enumeration item font Family-Face-Size
Sniff.EnumItem.Color:	2 0 92 92 92 0 #dark grey	Color for enumeration items. For format see WindowSystem.HighlightColor

ETRC entries for sniffgdb

Resource name	Default value	Description
sniffgdb.DebuggerExec:	gdb	Name of the debugger executable (must conform with the DebuggerAdaptor)
sniffgdb.DebuggerAdaptor:	Gdb4Adaptor	sniffgdb adaptor for the debugger backend (must conform with the DebuggerExec)
sniffgdb.DebuggerPrompt:	(gdb)	Prompt shown in the Debugger
sniffgdb.AddETSupport(Bool):	NO	Add support for the ET++ programming environment. Setting it to YES will add a new menu called Inspect that allows invocation of the ETPE from within sniffgdb.
sniffgdb.UseWordWrapForText(Bool):	NO	Wrap lines that are too long



INDEX

A

- about dialog, 175
- abstract classes
 - in Hierarchy Browser, 208
 - in Symbol Browser, 204
- AIX, 5
- analysis tools, 59
- applications
 - building, 9, 30
 - running, 16
- architecture, 169
- assignment filter, 212

B

- backup file, 263
- binaries, 7
- breakpoints, setting, 222
- build
 - clean, 17
 - definition, 7
 - environment variables, 12
 - examples, 13
 - generating, 28
 - global targets and rules, 11
 - log listing, 15
 - mistake, one target, 10
 - phases of, 8
 - process, 8
 - terminology, 7
- build tools, 23–35
- building projects, 7

C

- C++ templates, 205
- C/C++ syntax, 213
- call filter, 212
- cd, shortcuts, 95
- cdpath (environment variable), 95
- changing directories, shortcuts, 95
- Check in
 - in Editor, 215, 225
 - in Project Editor, 194
- Check out
 - in Editor, 215, 225
 - in Project Editor, 194
- Class Browser, 206
- Class menu, 180
- Class pop-up, 214, 225
- cleaning up after a test, 110
- client files, 7
- collecting timing information, 123
- colors, setting, 264, 265, 266
- combining
 - multiple TTest objects into single test, 114
 - operations into a single test class, 112
 - tests, 112
- commandline, 135
- commands, custom, 240
- comment, 216
- comparison Filter, 212
- compilation errors, jumping to, 228
- compiler options, 13
- copy, 182, 216, 226
- CopyInfo, identifying what test does, 115

copying files, 34

cp

See SmartCopy

CreateMake

definition, 24

syntax, 37–57

creating a test

dependencies on other tests, 114

requirements, 108

.cshrc

directory shortcuts, 95

Custom menu, 240

Cut, 182, 216, 226

D

debug target, 219, 228

debugger

See xcdb

commands in Editor, 222

custom menu, 240

declaration, switching to, 217

default text, 226

define parser configuration, 237

dependencies for make, 249

dependencies, creating tests with, 114

designing a test, 107

Directories menu, 173

directory, changing to, shortcuts, 95

Directory dialog, 174

Directory pop-up, 173, 175

Documentation Browser, 224

dragging text, in the Editor, 223

E

.e

See export file

Edit menu, 216, 226

editing shortcuts, 223

editing state, 214, 224

editor, 213

Class pop-up, 214, 225

custom menu, 240

dragging text, 223

editing state icon, 214, 224

Emacs, 213

fast copying, 223

find/changing, 217

list of symbols, 214, 225

matching brackets, 223

positioning, 217

text selection, 223

EditorCustomMenu, 240

Emacs, 250

configuring integration, 253

integrating, 251

working with, 252

emphasized text, 226

environment variables

build, 12

setting, 12–13

error

“Undefined symbol”, 25

Error log window, 177

ErrorFormats file, 241

ETRC file entries, 261

examining test results, 123

exception

Cleanup function, 110

handling, 123

stopping a test, 122

executables

building, 30

definition, 7

executing applications, 16

export file

definition, 7

generating, 27

extendable filters, 212

F

fast copying, 223

file copying, 34

File dialog, 172

File level symbol persistency, 243, 250

File list, 188

File menu

of the Editor, 214, 225

of the Project Editor, 188

files

- custom menu, 240
- directories & paths, 232
- ETRC, 239, 261
- makfile support files, 249
- project file, 239, 244
- Retriever filters, 240
- suffixes for, 233
- templates, 240
- used by SNIFF+, 239

Files menu, 173

Filter menu, 181, 212

filters

- extendable set of, 212, 240
- predefined
 - Assignment, 212
 - Call, 212
 - Comparison, 212
 - New, 212
- semantic, 210
- syntax for, 257

Find Error, 228

Find/Change Dialog, 171

FindSymbols, 25

fonts

- setting, 262
- setting in Emacs, 253

G

generate directory, 200

generating

- builds, 28
- executables, 30
- export files, 27
- libraries, 30

genproj, 246

grep, Retriever, 210

H

.h

- See* header file

handling exceptions, 123

header file, 7

heap corruption, 64

heap tools, 59

Hide overridden, 207

Hiding classes in the Hierarchy Browser, 209

Hierarchy menu, 209

Hierarchy Browser, 208

history

- menu, 181
- setting size, 264

History text, 196

I

Icon menu, 177

identifying what a test does, 115

ignore string

- parser configuration file, 237

Implementation

- switching to, 217

Info menu, 179

Information extractor, *see* sniffserver, parser

Inheritance relationship, 208

Inheritance Tree, 207

input

- parsing for test, 119
- test, 119

InterViews, 245

IPCpurge, 27

- See also* mop

K

key bindings in Emacs, 253

keyboard shortcuts, 183

L

Layout handle, 170
 libraries
 building from smaller libraries, 10
 generating, 30
 linking to export files, 27
 library projects, overlaying files, 234
 License dialog, 176
 line spacing, 262
 list of symbols, 204
 locking
 in Project editor, 192
 Locking menu, 194
 Project Editor, 192
 look, Motif, 233

M

macro parsing problems, 236
 make
 command, 234
 dependencies.incl, 249
 menu, 189, 219
 ofiles.incl, 250
 receiving options from Makeit, 11
 support for makefiles, 249
 See also Makeit, 10
 .Make, missing builds new makefile, 11
 MakeC++SharedLib, 30
 MakeExportList, 27
 makefile, 9–10
 description
 check in to RCS, 9
 naming convention, 9
 standard makefile, translating to, 9
 syntax, 9
 target types, 9
 standard makefile, creating, 9
 syntax, 9
 targets, 9
 update, 189, 218
 when to build, 11

Makeit
 definition, 28
 log listing, 15
 makefiles, when to build, 11
 passing options to make, 11
 MakeSharedApp, 30
 MakeShredLib, 30
 MakeSOL, 31
 matching brackets, 223
 Menu commands
 custom, 240
 shortcuts, 183
 modified icon, 214, 224
 mop, 31
 Motif look, 233
 multiple users, *see* locking

N

nest, 216
 new filter, 212

O

object file list for make, 250
 options
 compiler, 13
 overriding with variables, 13
 options, RunTest, 121
 overridden, hide, 207
 overriding, inherited MCollectible members of TTest, 110

P

parser
 configuration file, 239
 dealing with preprocessor macros, 236
 see also sniffserver
 parsing text inputs to test, 119
 paste, 182, 216, 226
 performing a test, 118
 persistency
 file level, 243, 250
 of SNIFF+ symbols, 243
 project level, 243

- .PinkMake, newer than *.Make, 11
- polymorphic testing, 118
- predefined filters, 212
- preferences, 231
 - colors, 264, 265, 266
 - ETRC, 261
 - fonts, 262
 - history size, 264
- Preferences dialog, 232
- Preprocessor macros, 236
- Print dialog, 175
- programs, building, 30
- Progress window, 176
- project
 - building, 10
 - building subprojects, 10
 - with many subprojects, 246
 - with separate implementation and declaration directories, 245
- Project Attributes dialog
 - for frozen subprojects, 203
 - for subprojects, 203
- Project Editor, 187
- Project file, 244
- project hierarchy
 - See* project
- project level symbol persistency, 243
- Project menu, 186, 189
- Project tree, 188
- protocol tests, 105
- providing input for test, 119
- purify, understanding messages from, 241

R

- RCS, 254, 192
- Redo, 182, 216, 226
- regular expressions syntax, 257
- resources, purging, 27
- results, examining test, 123
- Retriever, 210, 240
- reusable status, 170
- Run, 222
- RunDocument, 32
- running applications, 16

- RunTest
 - options, 121
 - overview, 117
 - run multiple tests, 112

S

- SCCS, 254, 192
- ScreamPlus, 31
- script, run multiple tests, 112
- search string, 210
- selecting text, 223
- semantic filtering, 210
- Setenv, 12
- setup, test framework, 110
- shared libraries
 - building, 9
 - definition, 7
 - generating, 30
 - linking to export files, 27
- SharedLibCache, 33
- shell tool, 227
- shortcuts
 - editing, 223
 - for menu commands, 183
- single stepping, 222
- site specific preferences, 231
- slcache
 - See* SharedLibCache
- Slibclean, 33
- SmartCopy, 34
- SNiFF+ locking, 255
- sniff-connect, 252
- sniff-disconnect, 252
- sniffserver, 235
 - running on a remote host, 236
 - see also* parser, 235
- source files
 - specifying in a project, 197
 - suffixes for, 233
- StartPink, 34
- status line, 170
- stopping, test, 122
- StopPink, 35
- streaming operators, test framework, 110

- subproject, 190
- subproject, building, 10
- suffixes, for sources files, 233
- super class, quick positioning to, 217
- Symbol Browser, 204
- Symbol list
 - in the Editor, 214, 225
 - in the Symbol Browser, 204
- Symbol table, 169
 - persistence of, 243
 - Update, 218
- syntax, for regular expressions, 257

T

- target
 - debug, 219, 228
 - make, 218, 228
 - name, 200
- teamwork, support for, 234
- template files, 240
- templates, C++, 205
- test
 - creating, 108
 - designing, 107
 - examining results, 123
 - identifying what test does, 115
 - input, 119
 - interface inherited from base class, 118
 - parsing input, 119
 - performing, 118
 - polymorphic, 118
 - stopping, 122
- test framework
 - class hierarchy, 104
 - cleanup, 110
 - collecting timing information, 123
 - combining multiple TTest objects, 114
 - combining operations in a single test, 112
 - combining tests, 112
 - example, 106
 - header files, 108
 - identifying test, 115
 - overriding MCollectible members, 110
 - overview, 103, 104
 - performing a test, 118
 - run test more than once, 110
 - script, 112
 - setup function, 110
 - test function, 109
 - tests with dependencies, 114
- test function, writing, 109
- this, 222
- timing information, collecting, 123
- tips and techniques, 95
- TLocalHeapAnalyzer, 62
- TLocalHeapMonitor, 61
- TMCollectibleTest, 105
- TTest
 - combining, 114
 - description, 104
 - hierarchy, 104
- TTestCollection, 105
- TTestMultiplexer
 - definition, 105
 - usage, 112
- TTextArgumentDictionary, test framework, 106
- TTieredText, test framework, 106
- TTieredTextBuffer
 - test framework, 106
 - writing text to console, 111
- TTimingTest, 105
- Tuning, 243
- Type Pop-up
 - Class Browser, 207
 - Symbol Browser, 205

U

- Undo, 182, 216, 226, 263
- Universal.Make, 11
- UNIX, shell interface, 227
- update Makefiles, 189, 218

V

- version control, 192, 254
- version file, 254

W

- window status
 - files, 242
 - preferences setting, 233
- working file, 193, 254
- Workspace Manager, 185
- writing a test function, 109
- writing a test to run more than once, 110
- writing text to the console, 111
- WYSIWYG, 213

X

- xcdb (debugger), 96
- xdb, 96
- xLC, wrapper for, 30

U94205-01A