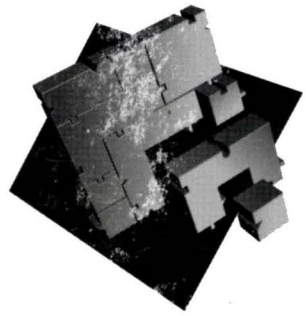
VERSION A FOR AIX®

Taligent®

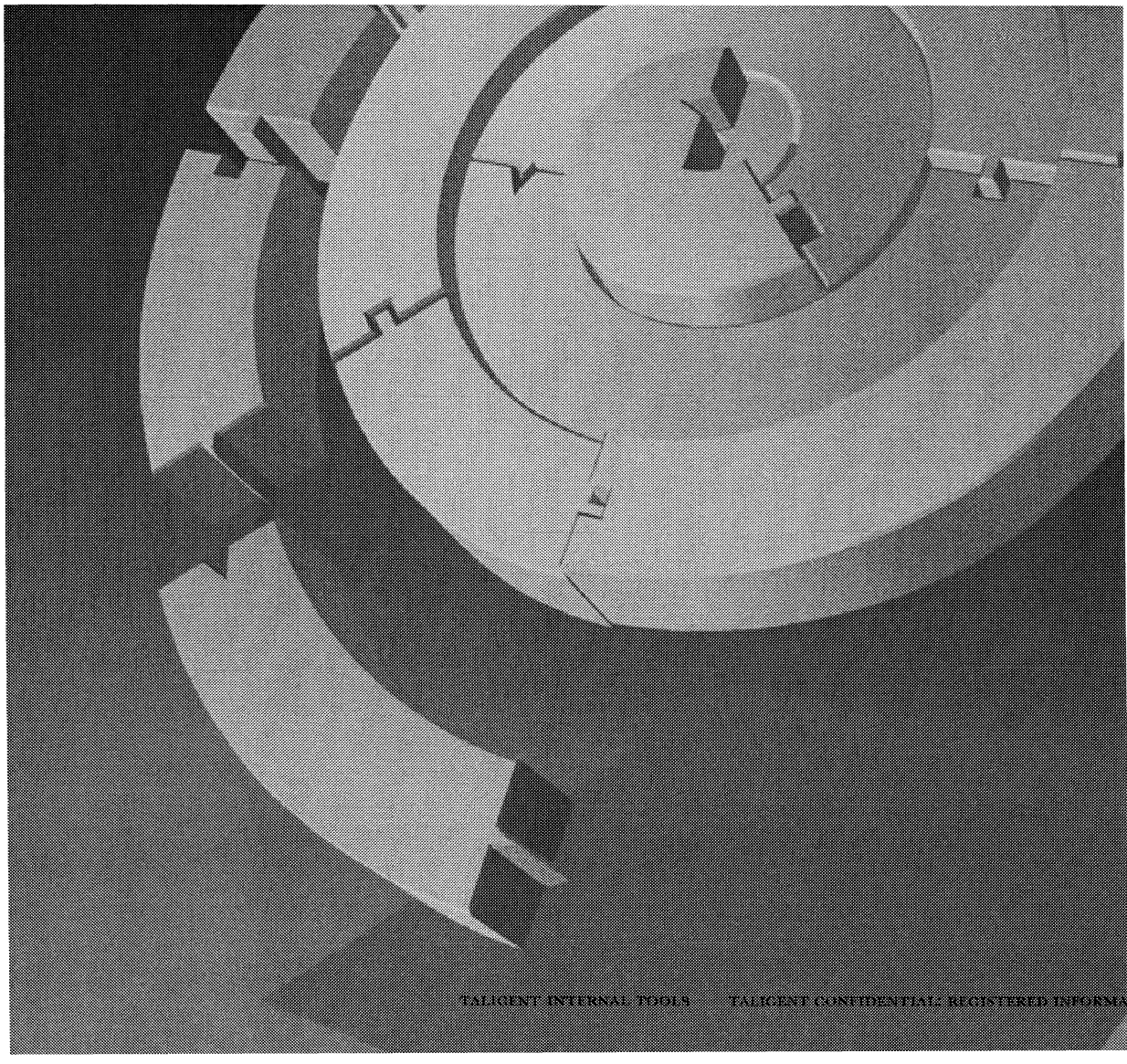
TALIGENT INTERNAL TOOLS



Taligent Internal Tools

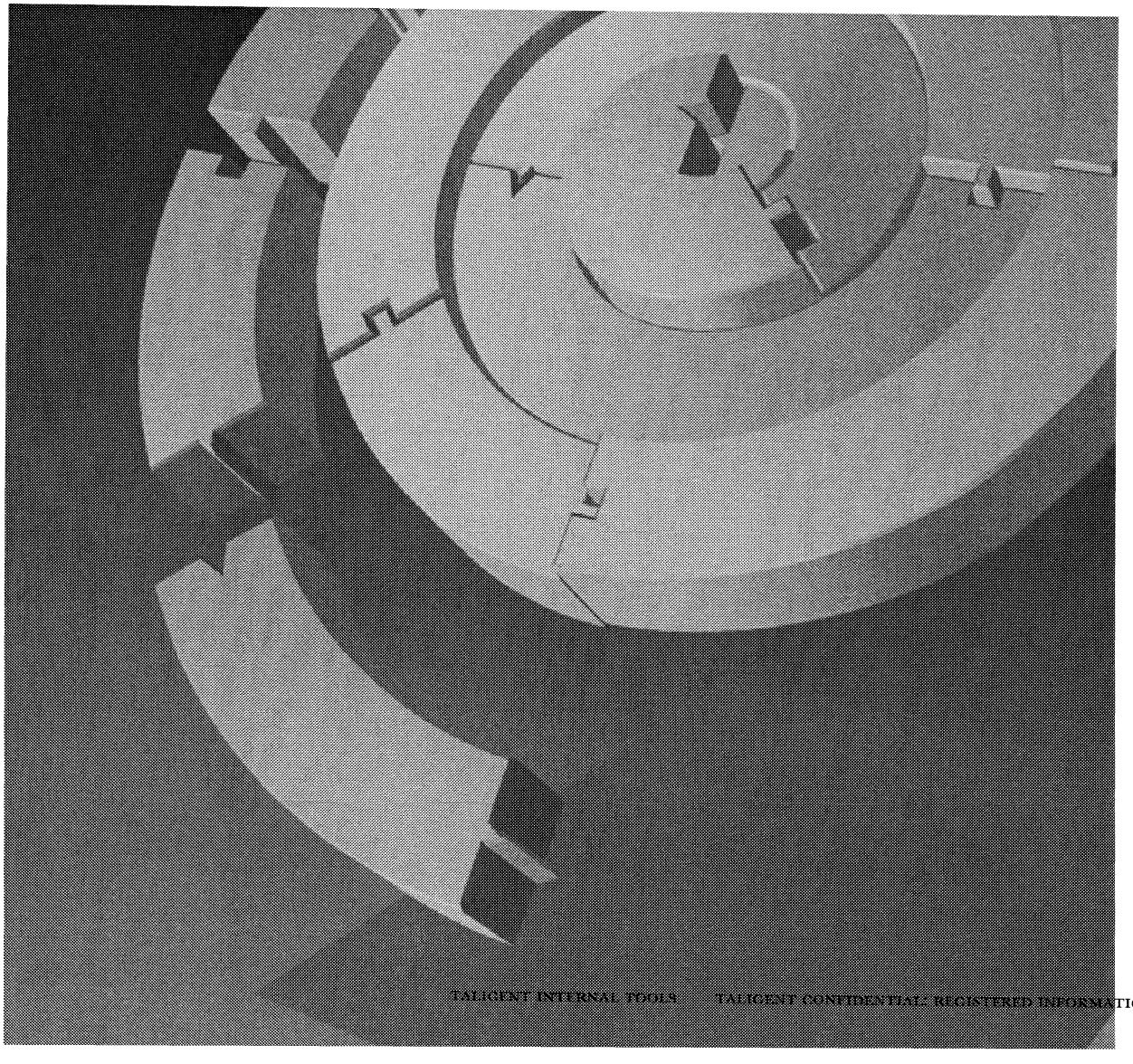
Taligent, Inc.
10201 North De Anza Boulevard
Cupertino, California 95014-2233
USA
(408) 255-2525

CONTENTS



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® and the Taligent Operating System.



A QUICK START

This summary, for internal Taligent developers only, is a quick overview to the topics of this book. It includes information on:

This summary is intended to quickly get you using the tools necessary to build Taligent systems and applications. This is *not* a substitute for the rest of this book, or for other more detailed company guides. To learn about the SNiFF+ programming environment, see the "Getting started" chapter in the *SNiFF+ Reference Guide* (Part III of *Taligent Tools for AIX).* Also, *The Methodologies and Processes Binder* (*The MAP*) explains the Taligent software development methodologies and processes.

GETTING STARTED

The instructions in this section will help you quickly set up your AIX environment so that you can start building your code. However, the instructions are terse with little or no explanation. For more detailed information on setting up and using your AIX environment, see "Working in the AIX environment" on page 3, which covers these steps in greater detail.

Setting up for the first time

1. Ask Technical Support to set up your AIX workstation with the Taligent standard setup and establish the appropriate server connections.
2. Install the default startup files. This will overwrite the `.cshrc`, `.profile`, `.mwmrc`, `.Xdefaults`, `.login`, `.xinitrc`, and `.emacs` files on your system.

   A **For layer work:**

   ```
   cd /usr/taligent/defaults
   InstallDefaults
   ```

   B **For native work:**

   ```
   /usr/taligent/defaults/NativeInit ~/Work
   ```

3. Log out completely and log in again to ensure proper execution of the new scripts. To log out:
   A. Choose End Session from the root menu by holding down the right mouse button on the desktop background.
   B. Choose OK.
4. **For layer work**, create a working directory. The working directory will become what is known as your $TaligentRoot. (Although you don't have to call your working directory *Work*, this is the Taligent standard name.)

   ```
   cd $HOME
   mkdir Work
   ```

5. Initialize the environment variables. The option `-l` indicates that you always want the latest build. The `-c` option creates all of the $TaligentRoot subdirectory trees on your local machine.

   A **For layer work,**

   ```
   SetRoot -l -c ~/Work
   ```

   B **For native work:**

   ```
   NativeRoot -l -c ~/Work
   ```

NOTE You need to run `SetRoot` or `NativeRoot` each time you log in to a terminal session that uses the Taligent build environment. If you get an error message like "### Command: Environment variable $TaligentRoot must be set!", it is because you didn't run `SetRoot` or `NativeRoot` in the session.

Installing builds

To install a build for the first time:

1. Verify that your machine has 400 Mbyte of free disk space.

```
df
```

2. Install the latest build:

   A. **For layer work,**

```
cd ~/Work
InterimInstall
SetRoot ~/Work
```

   B. **For native work:**

```
cd ~/Work
NativeInstall
NativeRoot ~/Work
```

Updating builds

To update to a later build

1. **For layer work,** (the `-b` first removes the existing build):

```
cd ~/Work
SetRoot ~/Work
InterimInstall -b
SetRoot ~/Work
```

2. **For native work:**

```
cd ~/Work
NativeRoot ~/Work
NativeInstall -b
NativeRoot ~/Work
```

Running layer programs

To run a program on the layer, first start the layer, and then run your program.

1. Start the layer:

```
cd $TaligentSharedLibs
StartPink
```

2. Run the Macrame program:

```
Macrame &
```

3. Quit the Layer:

```
cd $TaligentSharedLibs
StopPink
```

Running native programs

To run a Taligent Operating System program:

1. Transfer your program (*/home/mpogue/MyTest*) as binary to an Intel machine (*chrome*):

```
cd $TaligentSharedLibs
ftp chrome
type binary
cd /home/mpogue/MyTest
put Macrame
```

2. Start your program:

```
rlogin chrome
cd /home/mpogue/test
rp Macrame &
```

3. Stop your program:

   A. Run `jobs` to list running programs.

```
jobs
[1]  + Running rp
```

   B. Run `kill` to stop the program.

```
kill -9 %1
[1]    Terminated rp
```

Locating sample applications

The $TaligentSharedLibs directory contains libraries and sample applications. If you aren't already in the working directory, move there.

```
cd $TaligentSharedLibs
```

Here are three sample layer applications:

- To start the Mars application:

```
Mars documentName &
```

- To start the RunDocument Application:

```
RunDocument -c -o TTextStationery EditableTextLib&
```

- To start the Workspace Application (which brings up the Taligent Workspace Environment):

```
CreateWorkspace
LaunchWorkSpace
```

The source code repository

The source code repository is located in $TaligentSCMRoot. The directories on your local machine (created with `SCMCreateDirectories`) are parallel to the repository directories.

Taligent has a set of wrappers and extensions for accessing files in the repository. For example, when files are checked in or out, they are each associated with a specific build version number such as D31.1. Here is summary of the key commands and useful options to use when checking source in and out.

`Checkout -a -r`	Recursively checks out the latest build.
`Checkout -a -r -v D31.1`	Checks out all of the files in the directory that are in the D31.1 release.
`Checkout foo.C`	Checks out the latest `foo.C` file.
`Checkout -m foo.C`	Checks out the latest `foo.C` file for modification.
`NameVersions -f foo.C`	Display a list of all of the versions available for `foo.C`.
`Checkin -a -r`	Recursively checks in all of the files that are checked out.
`Checkin -i foo.C`	Checks in `foo.C` for the first time.
`Checkin -a -r -n D31.1`	Recursively check in all of the files that are checked out and set their build version to D31.1.
`CompareVersions D30.1 D31.1`	Compares what files have changed between build D30.1 and D31.1.
`CompareVersions -latest`	Shows what has changed in the current workspace directory compared with the latest in the repository.

LAYER SOURCE CODE EDITING AND BROWSING WITH SNiFF+

SNiFF+ provides a C/C++ development environment for browsing, cross-referencing, design visualization, documentation, editing, and debugging. SNiFF+ makes it possible to rapidly edit and browse large software systems in both a textual and graphical manner.

For more detailed information about SNiFF+, see the "Getting started" chapter in the *SNiFF+ Reference Guide* (Part III of *Taligent Tools for AIX).*

Installing SNiFF+

To use SNiFF+, you need to set three environment variables:

- In C Shell:

```
setenv SNIFF_DIR /usr/talilocal/packages/SNIFF
setenv LM_LICENSE_FILE $SNIFF_DIR/license.dat
```

  In your `.cshrc` file change the PATH variable to include:

```
$SNIFF_DIR/bin
```

- In Korn Shell or Bourne Shell:

```
SNIFF_DIR=/usr/talilocal/packages/SNIFF; export SNIFF_DIR
LM_LICENSE_FILE=$SNIFF_DIR/license.dat; export LM_LICENSE_FILE
```

  In your `.profile` file change the PATH variable to include:

```
$SNIFF_DIR/bin
```

Creating a project

To work in the SNiFF+ programming environment, you must have a SNiFF+ project. This can be done from inside SNiFF+ by following the instructions in "Creating a new project" in the *SNiFF+ Reference Guide* (Part III of *Taligent Tools for AIX).* Or, more easily, from outside of SNiFF+ with `genproj`, a command that creates a project consisting of all of the files in the specified directory, as well as creating subprojects in all of the corresponding subdirectories. `SourceDirectory` is your working directory, and `ProjectName` is the name you want to call the project. The `-e` indicates that subprojects should not be created in empty subdirectories.

```
genproj SourceDirectory -p ProjectName -e
```

NOTE To see all Taligent header files in your project, create a subproject to your project, and include in that subproject the header files in $TaligentIncludes (`$TaligentRoot/TaligentIncludes/Public`).

Editing in SNiFF+

SNiFF+ provides two choices for editing source code:

- SNiFF+'s own integrated editor (the default).
- An interface to standard emacs. Refer to "Emacs integration" in the *SNiFF+ Reference Guide* (Part III of *Taligent Tools for AIX)* to understand how to establish an interface between emacs and SNiFF+.

To edit source files:

1. Check out the layer sources you want to work on:

```
SetRoot ~/Work
SCMCreateDirectories
cd theDirectoryYouWantToWorkIn
Checkout -a -r
```

2. Start SNiFF+ with or without a project name. If you omit the project name, SNiFF+ loads an empty project. Starting SNiFF+ with a project name loads all of the source files, symbols and classes associated with that project for browsing and editing in SNiFF+'s editor window:

```
sniff MyProjectName &
```

Compiling and linking

The SNiFF+ programming environment currently works well for editing, browsing and debugging code. However, until SNiFF+ is integrated with the Taligent build tools, you need to compile and link projects in a UNIX shell. This shell can be either the Shell Window in SNiFF+ or a regular AIX shell window.

1. To compile and link:

```
SetRoot ~/Work
```

2. If the layer is running, stop it before executing `Makeit`:

```
cd $TaligentSharedLibs
StopPink
```

3. Run Makeit to build your project:

```
cd theDirectoryYouWantToWorkIn
Makeit
```

`Makeit` reads the `<project>.PinkMake` file and creates a makefile to compile and link the project. To understand the syntax of the PinkMake files, see "Makefiles" on page 43. You don't need to change the `*.PinkMake` file unless you add a new module to your project.

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.

Running a modified application

The linker, during `Makeit` execution, installs each compiled executable program in $TaligentSharedLibs. To run the application from an AIX shell window:

1. Restart the layer that was stopped for `Makeit`:

```
cd $TaligentSharedLibs
StartPink
```

2. Run your application:

```
YourApplicationName &
```

Debugging an application

SNiFF+, through its communication with either the `gdb` or `dbx` debugger, can be used to debug applications. Taligent 's specialized version of `gdb` is the default SNiFF+ debugger. The `gdb` executable is located in `/usr/local/bin/gdb`.

Before running the debugger in SNiFF+, set up the Project Editor Preferences:

1. Double click the project name in the bottom area of the SNiFF+ window. A Preferences dialog will appear.
2. Verify or setup the target (your application's name), the source path (the path to your source code), and the `Make` command (it should be `Makeit`).

To start the debugger from within SNiFF+, choose Debug Target target from the Exec menu of the Editor. This launches the debugger in a separate window from which you can set breakpoints, step through code, and print variable values.

If you don't want to debug within the SNiFF+ environment, use `xcdb`. This debugger brings up its own windowing environment in which to debug. Launch `xcdb` by using the Taligent script `xdb` along with the SourcePath option containing the path to your source.

```
xdb [-s SourcePath ... ] yourApplicationName
```

SYSTEM TESTS

Taligent uses three kinds of system tests: Basic Acceptance Tests (BATs), Subsystem Tests (SSTs) and System Test Applications.

BATs

To see what BATs are available and how to execute them:

```
cd $TaligentSharedLibs
RunBATS -h  gives help on running BATs
RunBATS -l  lists the available BATs
RunBATS <BATname>runs the specified BAT
```

The BAT source code is located in the `$TaligentRoot/Taligent/Testbed/BATS` directory structure.

SSTs

Numerous tests exist for the various subsystems of the layer. To install the prebuilt SSTs, use `InterimInstall` with the `-T` option.

```
InterimInstall -T
```

The SST test programs are in `<SSTtestname>/scripts` and `<SSTtestname>/bin` subdirectories within `$TaligentSharedLibs/Test/SST`. There is a wide variety of SSTs available that you should try. For example, an audio video test which executes a movie clip:

```
cd $TaligentSharedLibs/Test/SST/AVTests/scripts
AVTests.sh
```

The SST source code is located in the `$TaligentRoot/Taligent/Testbed/SubSystemTests` directory structure.

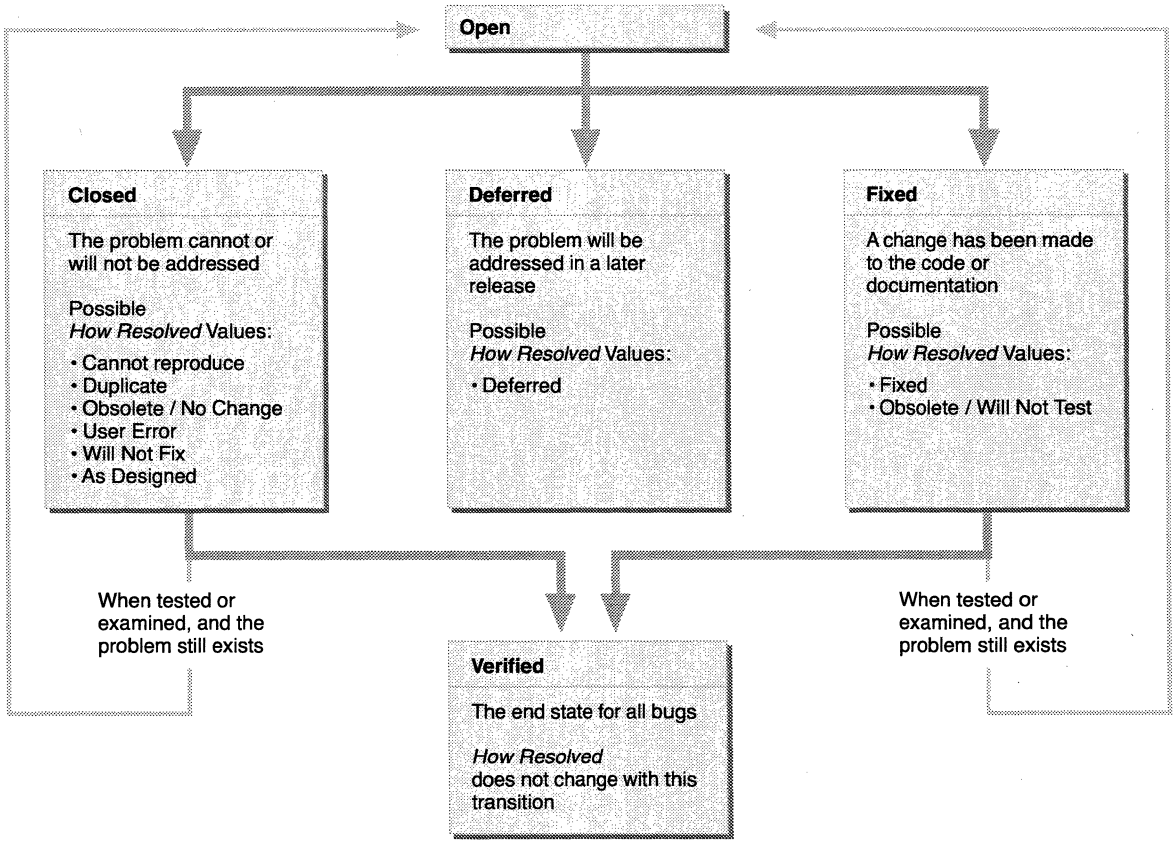
For a complete explanation of all of the SSTs locations, execution instructions and result interpretations, reference the Test Roadmap in `Central Services:Taligent Library:Test Doc Library:Test Doc:Tests`.

System test applications

The System Tests Applications should now be available, but information was not available when this book went to print. Please check with Product Test for information.

PROBLEM REPORTING

When you file a ProTeam problem report, the two key fields that indicate a problem report's state are *Status* and *How Resolved*. Here are the possible values for those fields and what they mean in a problem report's life cycle:



Terms and definitions

Here are the standard definitions used by Taligent Operating System for submitting ProTeam defect reports.

NOTE P0 and P1 defects are your highest priority work.

Priority field	P0 - Showstopper	Required for next Intermediate Build (Build will be held for fix.)
	P1 - Highest Priority	Required ASAP
	P2 - High Priority	Required for next Final D Build
	P3 - Medium Priority	Required for current release, must be fixed by Finish Phase Completion
	P4 - Lowest Priority	Not required for current release, fix only if time allows
Status field	Open	This is the default status for all new problem reports (either code or documentation). Open means action needs to be taken.
	Fixed	Code has been changed to repair the problem or feature. An ICBM notice has been submitted. The code has been checked in, and name revisioned.The *How Resolved* field should be set to "Fixed" or 'Obsolete/Will Not Test'. Remember, you must also fill in your ICBM number.
	Closed	There has been no change to code or documentation. *Closed* is used in conjunction with several of the "How Resolved" field choices, for example, Cannot Reproduce, Duplicate, Obsolete, User Error, and As Designed.
	Verified	The problem report has been verified in a master build and the "How Resolved" answer confirmed (such as "Yes it's fixed, yes there was User Error"). Fill in the Verified By and Verified Build fields.
	Deferred	This problem report will not be addressed in the current release. The Bug Priority Meeting (BPM) makes this decision. If a defect is deferred, its status will change to Open once the current release ships.
How Resolved field	Fixed	Code changes have been completed and the ICBM form submitted. Documentation corrections have been completed. Status also becomes Fixed
	Cannot Reproduce	The condition could not be recreated. Status becomes Closed.
	Duplicate	There's another report, or several reports detailing this same problem. You should fill in the duplicate problem report number or numbers in the Duplicate # field. Status becomes Closed.
	Obsolete/No Change	The code or documentation is now obsolete. No change to source code or documentation will occur. Status becomes Closed
	Obsolete/Will Not Test	The code or documentation is now obsolete. Code or documentation might have changed, but became obsolete before testing was performed. Status becomes Verified.

	User Error	The reporter of the defect has misinterpreted how the software functions. Status becomes Closed. Use this field in conjunction with the Status field. Additional Considerations Here: Does this user error indicate a problem in the documentation? Should TechComm be notified? If the answer is "yes", you have two choices: • Write a new problem report against the documentation • Change this report's Funct Area, Component, and Report Type to reflect a Documentation Error and change the status to Open.
	Will Not Fix	The problem will not be fixed for reasons noted in the description. Again, check the documentation for accuracy and clarity. Status becomes Closed.
	Deferred	The ICBM or Release Team has decided this problem report should not be addressed yet. Status becomes Deferred.
Severity definitions	Critical	The problem results in data loss or the corruption of data. There is no work-around to the problem and it is directly impeding the completion of work.
	Serious	The problem severely limits the use of the system or diminishes the functionality of the system. A work-around is available for problem allowing work to continue.
	Moderate	The problem limits the use of the system, but the majority of the necessary work can be completed. A work-around is available.
	Minor	The problem is annoying or unaesthetic, but is not a compromising problem.
Functional Areas and Components fields	Note that functional areas and components do not always map directly to products. Some components appear in several products, while others appear only in one particular product. Don't make any assumptions about products, when submitting defects.	
ICBM Scheduled Build field	Along with future builds, this field also lists future products so that you can indicate when you are deferring or delivering a response to the problem report.	
Product field	Indicates the product on which you found the bug. Any subsystem that is not targeted for a release, is organized under the Internal product.	

 NOTE If you have a bug that occurs on multiple source streams (layer and native, for example), submit the defect against the functional area and component in the layer responsible for the defect. For example, all Collections defects, even those that were found first on the Intel platform, need to go to:

Product = "Layer SDK1"
Functional Area = "Collections Text"
Component = "Collections"

If you have a bug that occurs on both native and OODDM builds, but the code is not there in the layer build, submit the defect against the native build. If you're not sure who to submit the defect against, ask your manager. There might be unusual cases that need to be handled differently.

NATIVE DEFECT AND CHANGE CONTROL PROCESS

Taligent Operating System uses a simplified version of the defect resolution and change process used by Taligent Application Environment. This section summary covers the steps to make and submit changes for the native Taligent Operating System build. This summary assumes that you are familiar with the basics of source control, and with the Taligent DIF development process.

NOTE Follow these steps carefully to ensure that your defect change is tracked properly. If you don't do it correctly, your change will probably get stuck somewhere, while the Build Team tries to figure out what you really meant to do. This could cause your fixes to be delayed!

Defect goes to OPEN state.

1. Submit a defect report, using ProTeam. This report normally takes less than 5 minutes to fill out.

   You need to submit a ProTeam defect for all defects, and problems. Soon the ICBM tickets will have ProTeam numbers, and vise-versa.

Defect goes to FIXED state.

2. Fix the defect. Open ProTeam, and change the state to *Fixed.* Save it.

   If the resolution of the defect requires a change to the build, then the resolver needs to fill out an ICBM ticket.

   A. Open the ICBM database, using the ICBM DB Opener. Filling it out gives you a ticket number, for example, *Native.1234.* ICBM tickets typically take about five minutes to fill out. See "Filling out ICBM forms" on page xxiv for more information.

NOTE The Build Team does not own any code. Engineers own PinkMake files, scripts, source code (`*.c`, `*.h`), and documentation (`*.d`). In some cases, there is old code in the build because many of files were simply moved over from the 68K build tree. If these files are in your functional area, then *you own them.* This ensures that the right thing is done with the files (deletion, .PinkMake modifications, renaming, and so on).

3. Use `NameVersions` on all affected files. This is not necessary if the files are being deleted. This way, only the `.PinkMake` will change. You *will* have to submit an ICBM ticket for the `.PinkMake` change.

4. Copy the new version numbers and filenames into the ICBM ticket.

   The ICBM ticket is then scheduled for a build. Normally this happens at the weekly Intel bringup/ICBM meeting (Thursdays, 1PM), but high priority ICBM tickets can be scheduled *when needed* upon request. Open ProTeam, and set the "Scheduled Build" field to the correct value.

Defect goes to VERIFIED state.

5. When you verify that the bug has been fixed in an actual build, open ProTeam and set the "How resolved", "Verified by:", "Actual Build", and "Verified Build" fields in the ProTeam bug report (see "Be sure to close the window, and log completely off the ICBM database, so others can use it efficiently—you must quit FileMaker Pro entirely." on page xxv). Change the state to "VERIFIED", and save it.

NOTE The assigned engineer is responsible for keeping the ICBM and ProTeam databases synchronized and up-to-date.

Filling out ICBM forms

To get the latest opener, see PacerForum `Tech Talk:TalAES Integration:ICBM`. Also see that forum for instructions on how to be notified about ICBM forms.

1. Open the ICBM database. When it asks for a password, just leave it blank, and click OK.
2. Create a new Change Notice with Cmd-N. The form will give you a unique revision string at the top of the form that looks like this:

   Native.*changeNumber*

   This is the name you use with `NameVersions` on your changed files.
3. Give the new Change Notice a meaningful title, for example, "PinkMake should not refer to foo.h".
4. Fill out the Submitter field with your name. Use the pop-up menu to ensure consistency. Select the proper Functional Area using its pop-up menu.
5. For Target Build, select "Native".
6. If architect approval is required, get it. Then, check the Architect Approved check box, and select the architect's name from the pop-up menu. Currently your architect in Taligent Operating System is Roger Webster.
7. Check the correct Change Classification, Change Type, and Client Impact. If these fields are confusing, ask your manager for clarification.
8. In the Fixed Bugs field, put the ProTeam defect numbers for all defects repaired by this change. There are very few cases where an ICBM ticket will *not* have an associated defect number. If you haven't already filed a defect for this change, do so now.

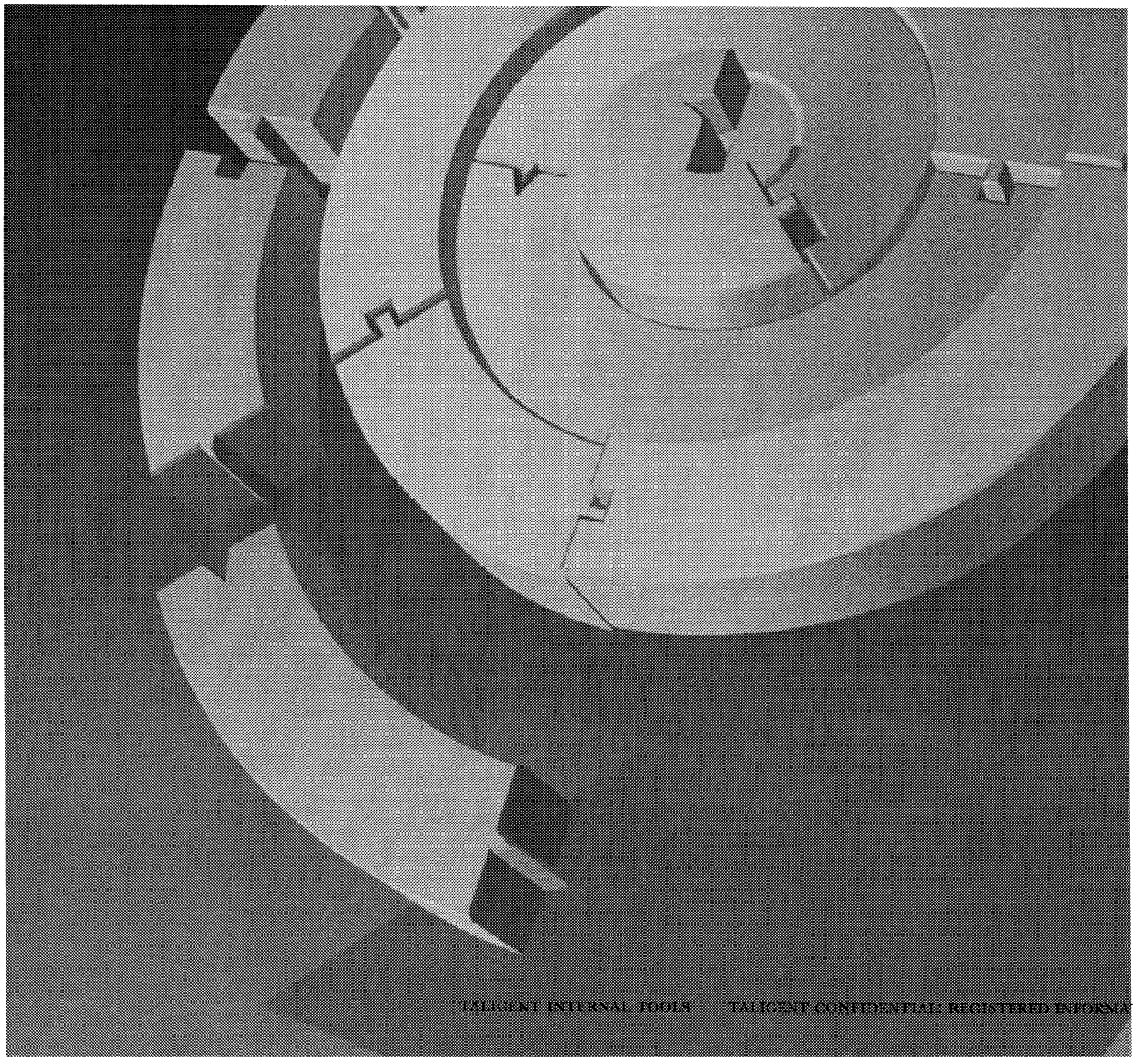
9 If your ICBM ticket requires that previously submitted ICBM's be integrated before *this* one can be integrated, put the dependent ICBM's number into the Dependent ICBM(s) field.

10 Check in the files associated with the change. Perform the name-revision (with `NameVersions`) on the affected files, using the string from the form.

11 Generate a list of all affected files, and put it into the "Generate ALL file paths..." field. Each filename must contain the SCM revision number associated with the file.

12 Go back to the Change Notice form and click the check box called "Form Completed and files name revisioned 'Native.XXXX'....".

Be sure to close the window, and log completely off the ICBM database, so others can use it efficiently—you must quit FileMaker Pro entirely.

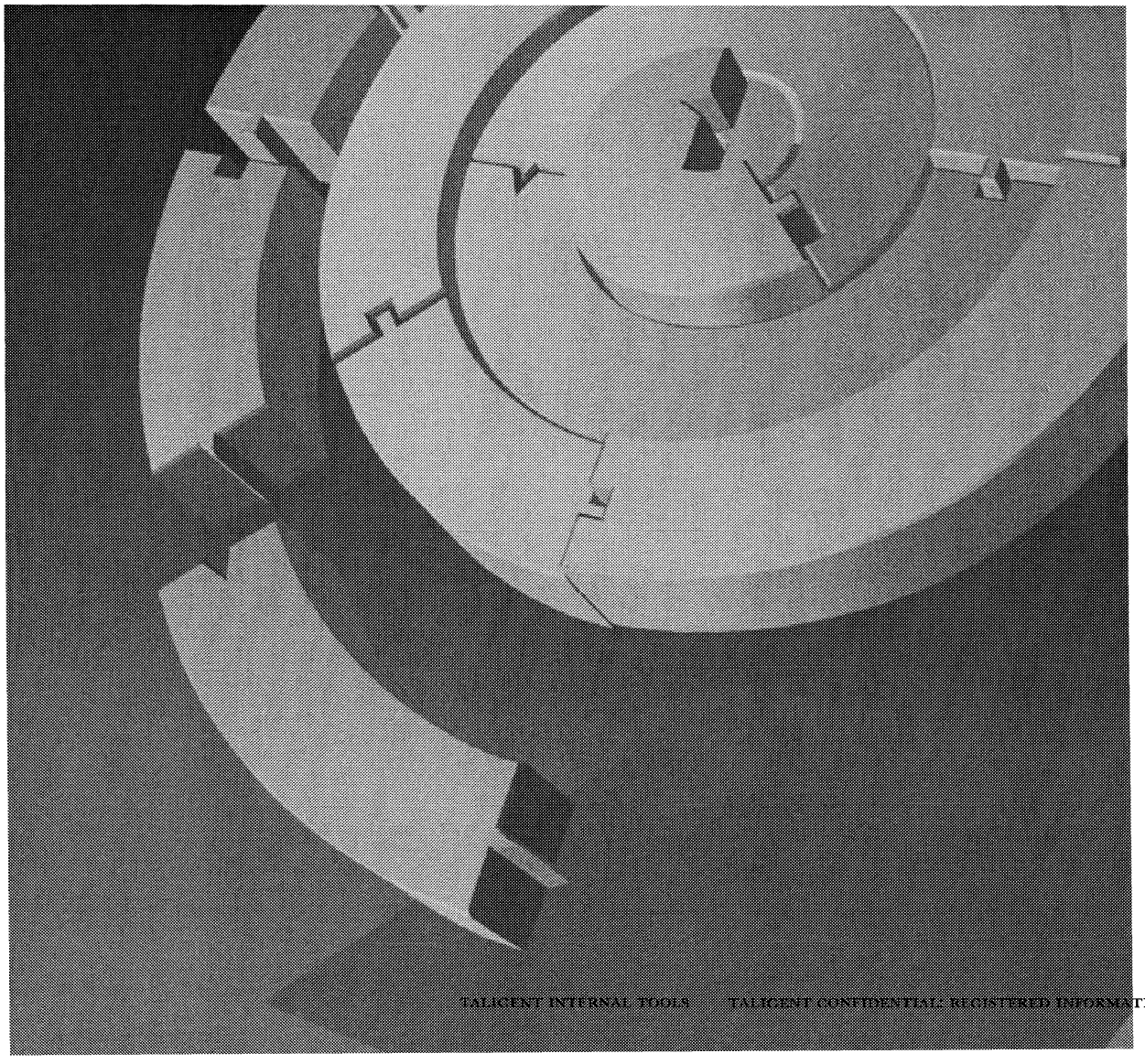


CHAPTER 1

INTRODUCTION

Taligent Tools for AIX describes the Taligent AIX development tools and how to use them. It also includes instructions for setting up your Advanced Interactive Executive (AIX) environment.

This guide assumes that Technical Support has installed the Taligent standard setup on your AIX workstation. It also 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.



CHAPTER 2

WORKING IN THE AIX ENVIRONMENT

Before you can build the Taligent Application Environment, the Taligent Operating System, or an application for it, you need to set up your AIX workstation. To build these applications, you need to know how to check source files in and out, how to build programs, and how to start and stop the layer or system.

To ensure that your environment will work with the Taligent AIX tools, you need to create a working environment compatible with the Source Code Management (SCM) and Build tools. For information about the SCM or Build environment, see the subsequent chapters.

SETTING UP FOR TALIGENT APPLICATION ENVIRONMENT

If you are working on or using the Taligent Application Environment, follow the setup instructions in this section. If you are working on or using the Taligent Operating System, follow the instructions in "Setting up for Taligent Operating System" on page 6.

Get the default startup scripts

Use `InstallDefaults` to copy the Taligent provided startup scripts (`.cshrc`, `.login.`, `.profile`, `.mwmrc`, `.Xdefaults.` and `.xinitrc`). These scripts were created by the Tech Support and Build teams and they set up initial values for various important shell variables related to building and running the layer, and then merge the old and new files. Once you have run this command, you should never need to run it again.

NOTE `InstallDefaults` overwrites the startup files already in your home directory. If you wish to save the information in your current startup files, rename your files before running the install script.

1. Run `InstallDefaults`.

   `/usr/taligent/defaults/InstallDefaults`

   `InstallDefaults` copies the files in the `/usr/taligent/defaults` directory (folder) to your home directory.

2. Completely log out and log in again in order to ensure proper execution of the new scripts. To log out:

   A. Choose End Session from the root menu by holding down the right mouse button on the desktop background.

   B. Choose OK.

Create your Work directory

Create the working directory where you want to install the Taligent Application Environment files. This directory name will eventually be the value of your $TaligentRoot shell variable.

1. Move to your home directory.

$home is a C Shell (`csh`) variable that contains the value of your home directory.

```
cd $home
```

2. Create the working directory. (You can use any legal AIX filename; however, this manual assumes that it is called Work.)

   ```
   mkdir Work
   ```

3. Create the root directory for your copy of the source code tree.

"~" is a csh shortcut that refers to your the home directory.

```
mkdir ~/Work/Taligent
```

CAUTION Do not create symbolic links from one directory to another, or from one file to another, in the repository or in your workspace. Problems can occur because the tools see these links as two separate directories.

Initialize your environment

Initialize your environment with `SetRoot`. `SetRoot` sets the value of $TaligentRoot and several related shell variables. $TaligentRoot is the directory from which all of your Taligent Application Environment directories and files descend.

1. Run `SetRoot` and specify your working directory. Optionally, you can include `-O` to specify Optimize during compilation.

```
SetRoot -l ~/Work
```

NOTE You need to run `SetRoot` each time you login to a terminal session that uses the Taligent build environment. If you get an error message like "### Command: Environment variable $TaligentRoot must be set!", it is because you didn't run `SetRoot` in the session.

Never set $TaligentRoot directly—use `SetRoot` instead because it also sets other important related variables. For more information, see "SetRoot" on page 38.

Create your copy of source tree

To work with source files and use the SCM tools, you must set your source tree to mirror the directory structure in the SCM repository. Use `SCMCreateDirectories` to create the directory structure in your environment.

1. Move to your Taligent directory.

```
cd $TaligentRoot/Taligent
```

2. Run `SCMCreateDirectories`. It might take a few minutes to complete.

```
SCMCreateDirectories
```

For more information, see "SCMCreateDirectories" on page 32 and Appendix B, "Taligent source code maintenance" on page 171.

Install the build

To install the current Taligent Application Environment build, run `InterimInstall`. `InterimInstall` is a temporary script that installs a build into your Work directory.

1. Move to your Work directory in your working directory.

```
cd $TaligentRoot
```

2. Run `InterimInstall`. It might take a few minutes to complete.

```
InterimInstall
```

For more information, including how to specify particular builds to install, see "InterimInstall" on page 64.

SETTING UP FOR TALIGENT OPERATING SYSTEM

If you are working on the Taligent Operating System, follow the setup instructions in this section. If you are working on the Taligent Application Environment, follow the instructions in "Setting up for Taligent Application Environment" on page 3.

Prepare your environment

Use `NativeInit` to set up your entire AIX environment and prepare it for building Taligent Operating System code. Once you have run this command, you should never need to run it again. `NativeInit`

- Sets up the `.cshrc`, `.login`, `.profile`, `.mwmrc`, `.Xdefaults`, `.xinitrc`, and `.TaligentStartup` files. These scripts, created by the Tech Support and Build teams, set up initial values for various important shell variables related to building and running the layer, and then merge the old and new files.
- Sets your environment variables correctly.
- Creates a directory tree (wherever you want it) to hold Taligent Operating System source code and binaries. This tree is called your *workspace.*

NOTE `NativeInit` will request to backup your existing startup scripts (`.cshrc`, etc.). The default is to save copies of your files.

To prepare your environment:

1. Run `NativeInit` and specify the absolute path for your workspace. If the workspace directory doesn't exist, `NativeInit` creates one for you.

   ```
   /usr/taligent/defaults/NativeInit ~/Work
   ```

2. Log out completely and log in again in order to ensure proper execution of the new scripts. To log out:
   - A. Choose End Session from the root menu by holding down the right mouse button on the desktop background.
   - B. Choose OK.

Install the Native TalOS build

To install the current Taligent Operating System build into the directory structure you have just created, run `NativeInstall`. This script installs binaries, libraries, and tools into the proper places in your workspace (install source code separately with `Checkout`).

1. Move to your Work directory.

```
cd $TaligentRoot
```

2. Run `NativeInstall`, it might take a few minutes to complete. For this transition, the release name is N10.1.

```
NativeInstall -b -r N10.1
```

`NativeInstall` automatically retrieves the tools that the specified build requires, and installs them in the correct $TaligentRoot/ToolsDir. Unlike the layer build environment, native tools are always synchronized with native source code releases to help ensure correct builds. You can override tool installation with the `NativeInstall -T` option.

CAUTION Do not create symbolic links from one directory to another, or from one file to another, in the repository or in your workspace. Problems can occur because the tools see these links as two separate directories.

NOTE Do not use `InterimInstall` (for Taligent Application Environment) and `NativeInstall` to install in the same workspace.

If you want to be able to switch between the two build environments, know that:

- `NativeInit` *directoryName* performs a `NativeRoot` *directoryName* as part of the installation procedure. `NativeRoot` is equivalent to the `SetRoot` command used in the layer environment.
- You can switch between build environments, as long as you always execute `NativeRoot` *directoryName* or `SetRoot` *directoryName* first. Be careful, because these commands change `.TaligentStartup` in your home directory. Because of this, you can't easily have a simultaneous native and layer build.

BUILDING PROJECTS

For details about building projects, see Chapter 4, "The build environment."

CHECKING FILES IN AND OUT

Once you have set up your environment to work with the SCM and Build tools, you can check source files in or out of the SCM version control database. To check files in or out, use the `Checkin` and `Checkout` commands. These and other SCM tools are documented in Chapter 3, "Taligent SCM tools." For information about the SCM database, see Appendix B, "Taligent source code maintenance."

Change your working directory first

Before using `Checkin` or `Checkout`, you must move to the directory in your workspace to which the corresponding files will be checked out. This directory corresponds to the directory in the project hierarchy where the source file resides. For example, to check out the files from the HeapTool project, change your current directory accordingly:

```
cd $TaligentRoot/Taligent/Instrumentation/HeapTool
```

Checkout

Checkout retrieves files from the SCM directory hierarchy and puts them into your directory hierarchy—your working directory. For information about `Checkout`, see "Checkout" on page 22.

Examples

Check out read-only copies of the latest versions of the specified files:

```
Checkout file1 file2 ...
```

Check out modifiable (`-m`) copies of all (`-a`) the files in the project directory:

```
Checkout -m -a
```

Check out versions of all files corresponding to the symbolic name *D4Release:*

```
Checkout -v D4Release -a
```

Check out all the files in the project directory that have the symbolic name *d32.1_Final.* The –r option tells `Checkout` to operate recursively in all subdirectories in the project and performs the same `Checkout` in each. This example gets the sources for a particular build:

```
Checkout -v d32.1_Final -a -r
```

New files

In order for the SCM files to work properly, source files require a comment near the beginning of the file that contains either `"$Revision:$"` or `"$Header:$"`. The SCM tools store the file's version information in this string, and update it every time you check the file out.

Files you check out should already contain one of these two magic strings. You need to include one of these strings when checking in a file new to the project. When you use `Checkin -i` to check in a new file, it calls `SCMInsertHeader` to insert the string for you. For more information, see "SCMInsertHeader" on page 36.

Checkin

`Checkin` submits your changed files into the SCM repository. For information about `Checkin`, see "Checkin" on page 19.

Examples

Initialize the file. Use this when a file is not already in the project. This command checks in the first version of the file. If your working directory has no corresponding directory in the project, you get an error:

```
Checkin -i file1
```

Initialize the file, and create a project directory if one does not already exist that corresponds to this working directory. This is useful when first checking a whole subtree into the project:

```
Checkin -I file1
```

Check in all files and immediately check them out for modification:

```
Checkin -m -a
```

Check in files and designate the newly checked-in versions with the symbolic name *ap_latest*, even if another version of the file is already designated with that name (see "NameVersions" on page 28 for information about symbolic names):

```
Checkin -N ap_latest file1 file2 ...
```

Branching

To create a branch, check a version out for modification, then check it in again; `Checkout` does not actually create the branch—`Checkin` does. The method for branching depends on whether or not the version you are branching from is the highest version on the trunk or branch.

Making a branch

To branch from version 1.27 of a file, when version 1.27 is *not* the last version on the trunk, use:

```
Checkout -m -v 1.27 file        # Gets version 1.27 for modification.
Checkin -f file                 # Creates a branch, 1.27.1.1.
```

The result is that you have created a branch, and the file you will have in your workspace is version 1.27.1.1.

If the version you want to branch from is the highest-numbered version on its trunk or the branch (such as version 1.30):

```
Checkout -m file                # Gets version 1.30, the top of the trunk
Checkin -f -v 1.30.1.1 file     # Creates a branch
SCMAdmin -u -v 1.30 file        # Cancels your lock on 1.30
```

Because you acquired a lock on version 1.30, and then checked in version 1.30.1.1, you must do the extra step of releasing your lock on version 1.30, which is what `SCMAdmin` does.

Using a branch

After creating a branch, you can perform all the normal functions, such as designating versions on the branch with symbolic names using `NameVersions`. Those versions can be retrieved when you check out by name, use `SyncSources`, and so on. You can check out versions for modification and check them back in, but you have to do so carefully. If you use `Checkout` only, you always get the highest-numbered version on the trunk. To get a version from a branch you must specify the version number or use a symbolic name:

```
Checkout -v 1.30.1.5 file
```

When you check out a branch version for modification, you can use `Checkin` without any special arguments to check that version in again. The new version will be checked in to the branch, and that version will be checked out again as a read-only file in your workspace.

Naming a branch

One good way to work on a branch is to name it. When you create the branch, pick a name like JohnsBranch to designate each new version on that branch. Use that name every time you check out the file (or a set of files you have branched). Also, every time you check in new versions of the files you have branched, designate the new version with that name. That way the name always designates the latest version on the branch you are working on. For example, assuming that the latest version of WorkFile.C is 1.17:

```
Checkout -m WorkFile.C
<edit the file>
Checkin -v 1.17.1.1 -n JohnsBranch WorkFile.C
```

This creates the branch. To later modify your branched version of the file:

```
Checkout -v JohnsBranch -m WorkFile.C
<edit the file>
Checkin -N JohnsBranch WorkFile.C
```

Note the capital 'N'.

This checks out the latest version on the branch and checks in a new version, carrying the name JohnsBranch along to the new version. Use the uppercase `-N` because the name JohnsBranch already designates one version of the file, and you want to change the version this existing name refers to.

Class and member descriptions

Check in class and member description files (*.d) in a Docs subdirectory. For example, if you have a header Foo.h that resides in `$TaligentRoot/Taligent/Platform/AIX` then the corresponding description file Foo.d resides in `.../Platform/AIX/Docs`.

NOTE For native builds in the MPW build environment, class and member description files (`*.d`) were usually kept in the source code directory itself. Because the AIX build environment uses automated tools to grab the files and format them, be sure to keep all `*.d` files in the `/Docs` subdirectories.

Other SCM tools

In addition to `Checkin` and `Checkout`, other useful SCM tools and scripts include:

- `SyncSources`—an optimized `Checkout` that does not check out those files for which you already have the correct version in your workspace.
- `CompareVersions`—displays the difference between a file in your workspace against a file in the project. See page 24.
- `ListVersions`—reports the workspace version of each file in the current directory. See page 26.
- `NameVersions`—associates a symbolic name with a set of files in an SCM project or project hierarchy. See page 28.

STARTING AND STOPPING THE TALIGENT APPLICATION ENVIRONMENT

To run a Taligent Application Environment program on AIX, you must currently execute the program on the Taligent AIX reference layer.

Starting the layer

To start the layer, run `StartPink`.

1. Move to the directory containing `StartPink`.

```
cd $TaligentSharedLibs
```

2. Run `StartPink`.

```
StartPink
```

For more information, see "StartPink" on page 75.

If `StartPink` is successful, you can start an application. For example, to run Macrame or SimpleText:

The "&" runs the command in the background.

```
Macrame &
SimpleText &
```

Stopping the layer

Run `StopPink` to safely stop the layer. For more information, see "StopPink" on page 75.

1. Run `StopPink` to safely stop the layer.

```
StopPink
```

To restart the layer, rerun `StartPink`.

STARTING AND STOPPING TALIGENT OPERATING SYSTEM PROGRAMS

To run a Taligent Operating System program, you must transfer your program to an Intel-based computer, and explicitly start the program and system.

Transferring your program

To transfer your program to an Intel machine, use `ftp`.

1. Move to the AIX directory containing your program.

```
cd $TaligentSharedLibs
```

2. Run `ftp`. In this example, the target machine is *chrome.*Enter your password, when requested.

```
ftp chrome
```

3. Set the transfer type to binary.

```
type binary
```

4. Optionally, turn transfer feedback on to print "#" for every block transferred.

```
hash
```

5. Change your `ftp` working directory on the remote machine to the directory where you want to put the Intel binary.

```
cd /home/mpogue/test
```

6. Copy the file from your AIX workstation to the Intel machine.

```
put Macrame
```

Starting your program

To execute on the Intel machine, use `rlogin` to remotely login.

1. Run `rlogin` from your AIX workstation. Enter your password when requested.

```
rlogin chrome
```

2. On the remote machine, change to the directory containing the binary image you transferred.

```
cd /home/mpogue/test
```

3. Run `rp` to start your program. (`rp` replaces the `runpink` program previously used to load and run programs.)

The "&" runs the program in the background.

```
rp Macrame &
```

NOTE To debug your program, use `gdb`. For information about `gdb`, see Chapter 10, "GDB" on page 149.

Stopping your program

Use the UNIX `kill` command to safely stop your program.

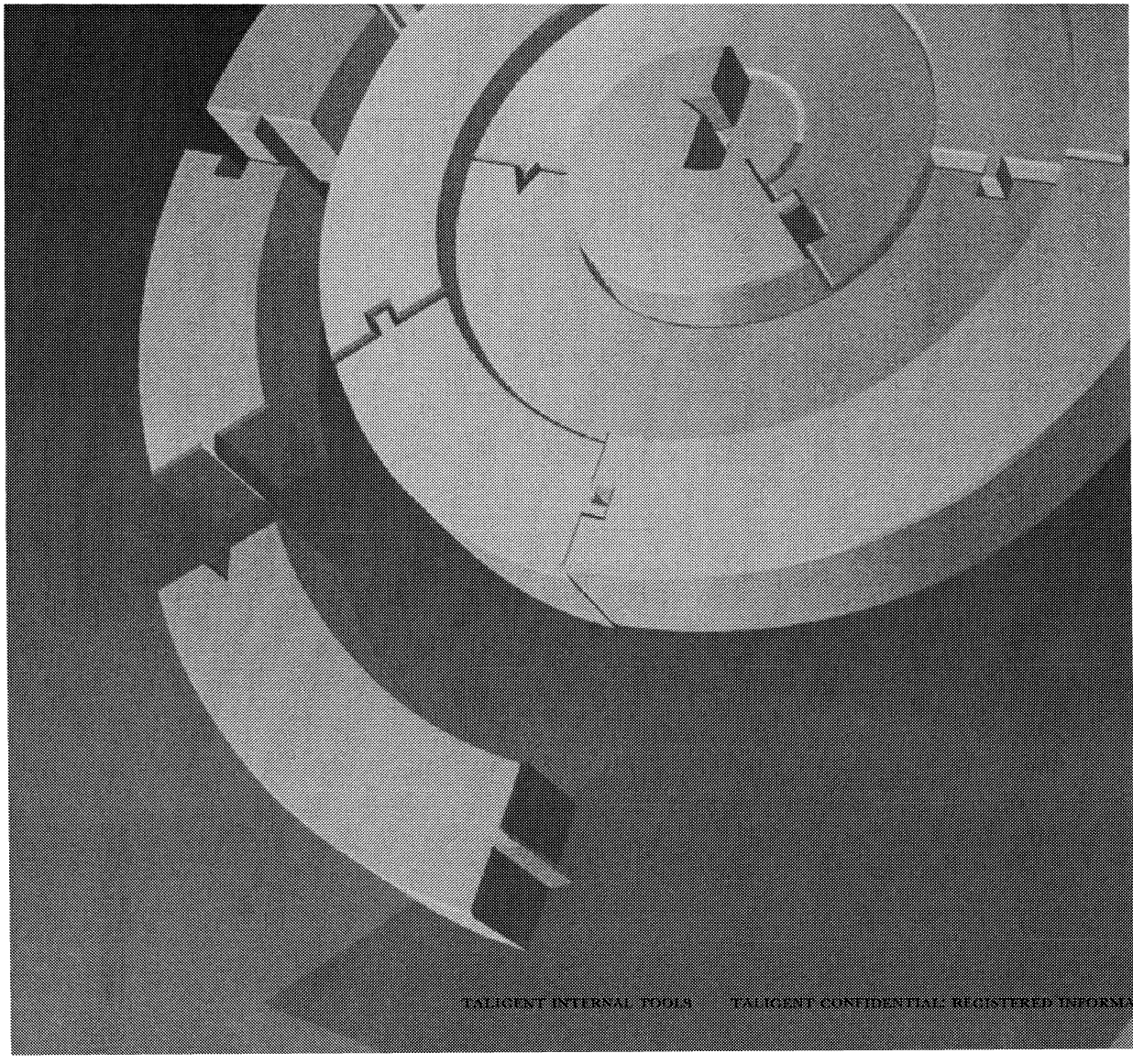
1. Run `jobs` to list running programs.

```
jobs
[1]  + Running rp
```

1. Run `kill` to stop the program.

%1 kills the [1] program

```
kill -9 %1
[1]    Terminated rp
```



CHAPTER 3

TALIGENT SCM TOOLS

Before you start to use the Source Code Management (SCM) tools, run `SetRoot` (layer) or `NativeRoot` (native) to set the environment variables that these tools depend on. Also make sure you have created a mirror of the SCM project by using `SCMCreateDirectories`. If you have not done this, follow the instructions in "Setting up for Taligent Application Environment" on page 3.

Most of the SCM tools assume you are in the working directory of interest before running the tool. For example, before working on the Albert project, change to the Albert directory.

```
cd $TaligentRoot/Taligent/Portable/AES/Albert
```

Then, check out all the files in the Albert project from one consistent build.

```
Checkout -a -r -v D32.29
```

At this point, you have all the source files for the Albert project that were checked out from their home in the SCM hierarchy.

NOTE Each user has a private snapshot of the system. When you build a project (or project hierarchy), *everything* is on the local file system— header files, export files, and executables. This is your *workspace.*

The only way other that people can see your changes is if you check in your changes using `Checkin` and `NameVersions`. Others can then see your changes in the next system build, or when they directly check out and build your project.

NOTE All Taligent tools require that the *filename* argument be the last argument on the command line; all options must precede *filename.*

SYMBOLIC NAMES

The system of symbolic names (used by `NameVersions` and the other tools) is implemented using a file called `names` in a subdirectory called `.TaligentSCM` (note the initial period) that is created in the repository and in your workspace. There is one names file per directory in the repository. Normally this is invisible to you. However, there are several considerations that you should be aware of:

- The `names` file in the repository is the *final word* about symbolic names for that directory. Files in workspaces are local copies of the file in the repository. If the local file's last-modified date and time are newer than the file in the repository, then its contents are used. If the file's date and time are older, a fresh copy is checked out. This is all done internally—you never see it. However, this means that the clocks of the machines on the network should be synchronized, or at least very close.
- The `names` files are controlled by SCM, just like your source files. This prevents corruption that can occur if two people run `NameVersions` at the same time in the same directory. If two people do step on each other, one of them gets a message that the "`names` file could not be checked out." If this happens, just rerun `NameVersions`.
- Because of the locking mechanism used by SCM, two `NameVersions` processes run by users *with the same name* can cause corruption of the `names` files. This can happen if you run `NameVersions` twice in parallel on a single machine, or anywhere on the network, using the same login name.

CAUTION Do not run two `NameVersions` commands anywhere on the net at the same time with the same user ID.

CHECKIN

`Checkin` submits your files into the SCM repository. Run `Checkin` for all project files that you checked out for modification.

Syntax

```
Checkin [-n Name | -N Name] [-i | -I [-b] ] [-v version] [-a] [-C] [-f] [-m]
        [-r] [-q] [-D] [filename...]
```

Arguments

–a	*all:* Check in all files in the workspace directory into the corresponding project directory. Use –a in place of *filename*.
–b	*binary:* Declare the checked in files as binary. Suppresses header substitution on the magic strings during checkout. Only works with –i and –I.
–C	*Comment:* Suppress comments strings for all files that are checked in; all files will have empty comment strings for that version. `Checkin` does not read standard input.
–D	*debug.* Include debug information in the output for debugging `Checkin`.
–f	*force:* Force `Checkin` to use a new version, even if the files are unchanged.
–i	*initialize:* Initializes a new file (the first version) in the project directory when a file is not already in the project. An error occurs if your working directory has no corresponding project directory.
–I	*Initialize:* Initializes a new file *and* creates a project directory if none correspond to this working directory. Note: the parent of the working directory must exist in the project.
–m	*modify:* After checking the file in, check it out for modification.
–n *Name*	Check in the files and designate the symbolic name *Name* as the new version. See "NameVersions" on page 28 for more information about symbolic names.
–N *Name*	Check in files and designate the symbolic name *Name* as the new version, even if another version of the file already has that name.
–q	*quiet:* Suppress commentary (but still report errors).
–r	*recursive:* Run this `Checkin` command in this directory and recursively down the subdirectories in the workspace.
–v *version*	*version:* Specifies a particular version of each file. *version* can be a version number (like 1.4).
filename	The name of file in the corresponding project directory. Separate multiple filenames with white space. Use –a when you want all the files in the project.

Usage

Before using `Checkin`, change to the directory in your workspace that contains the file you plan on checking in. For example, to check in the files from the Tokens project, change to your corresponding Tokens directory:

```
cd $TaligentRoot/Taligent/Portable/OES/Tokens
```

After checking in the file, `Checkin` retrieves a read-only copy for you. If you want to keep your lock on the file, use –m to check the file in, and then immediately check out again for modification.

```
Checkin -m file1.C
```

Comments

`Checkin` prompts for a comment that applies to all files that are checked in. `Checkin` reads the comment from standard input so you can redirect to it from a file. If you want a separate comment for each file, run `Checkin` separately for each.

NOTE The `Checkin` prompt instructs you to finish your comment by typing a single period or Ctrl-D. Be sure to avoid the common mistake of pressing Return and endlessly waiting for a new prompt.

Messages

After `Checkin` submits a file, it displays a message indicating the file's status and new version. The three file-status messages are:

- checkin—a normal check in
- new—a new file
- revert—the file *reverted* to the previous version because the file is identical to that version

When you use –a to check in all files in the current project, `Checkin` prints a warning for files not checked out, but continues checking in the rest of the files.

```
# enter log message, terminate with single '.' or CTRL-D (end of file)
This is the user-entered comment text.
.
# Checkin: ERROR: foo.C is NOT checked out for modification by arn
# Checkin: ERROR: bar.C is NOT checked out for modification by arn
checkin file.C,1.10
```

Prompt for comment — *Comment* — *End the comment* — *Not checked in* — *Not checked in* — *Normal check in*

For recursive check ins that use –r, the listing looks the same, except that there is an additional message for each project that it traverses.

```
# recursively checking in for /home/…/Toolbox/Tokens...
```

If you attempt to check in a file that has never been checked in, `Checkin` displays:

```
### Checkin: WARNING: "file.C" is not part of the current project
### if "file.C" is a new file, use the -i/-I option with Checkin
```

As the message says, you should include –i to indicate an initial version of the file.

New files or projects

To add a new project to the existing hierarchy, such as TestWindowServer, a new sub-project of the WindowServer project:

1. Create the directory on your local file system if you have not done so already.
2. Copy all the source files and TestWindowServer.PinkMake into the directory.
3. Move to that directory and check in the files. Include –a to check in everything, and –I to create the project directory and initialize the files.

Checkin *messages*

```
% Checkin -I -a
# Checkin - Creating rcs dir "/Repository/tools/Checkin/test"...
new foo.C,1.1
new bar.C,1.1
```

Examples

Check in files and designate the symbolic name *ReadyForBuild* as the newly checked-in version:

```
Checkin -n ReadyForBuild file1.C
```

Check in files and designate the symbolic name *D34.FINAL* as the newly checked-in version, even if another version of the file is already designated with that name:

```
Checkin -N D34.FINAL files...
```

Force a new version to be checked in even for files that have not changed (otherwise unchanged files revert to the previous version):

```
Checkin -f -N D35.FINAL files...
```

Check in all the files in the current directory and immediately check them out for modification:

```
Checkin -m -a
```

CHECKOUT

`Checkout` retrieves files from the SCM directory hierarchy (the source code databases) and puts them into your directory hierarchy—your working directory.

Syntax

```
Checkout [-a] [-m] [-c] [-r] [-o outFile] [-q] [-D] {-v version… | -latest} filename…
```

You must specify either `-latest` or `-v`.

Arguments

–a	*all:* Check out all files from the corresponding project directory into the workspace directory. Use –a in place of [*filename*...].
–c	*cancel:* Cancel the check out of files checked out for modify.
–D	*debug:* Include debug information in the output.
–latest	Check out the highest numbered version on the trunk. You must specify either `-latest` or `-v`.
–m	*modify:* Check out the files for modification. Only one person can have a particular version of a file checked out for modification.
–o *outFile*	*output:* Write the checked-out file to *outFile* instead of its own name. Use this to make a temporary copy of some version of a file without disturbing the copy in your workspace. For example, to get version 1.5 of Bundles.C and save it to BundlesTemp: `Checkout -v1.5 -o BundlesTemp Bundles.C` You cannot use –o with –m, –p, –r, –a, –c, or with more than one *filename*.
–p	Write the checked out file to `stdout` instead of a file on disk. You cannot use –p with –m, –o, –r, –a, –c, or with more than one *filename*.
–q	*quiet:* Suppress commentary (but still report errors).
–r	*recursive:* run this `Checkout` command in this directory and recursively down the subdirectories in the project.
–v *version*	*version:* Specifies a particular version of each file. *version* can be a version number (like 1.4) or a symbolic name (see "NameVersions" on page 28). If you specify multiple –v arguments, `Checkout` behaves as if you gave multiple commands, one for each symbolic name (version), in the order given. You must specify either `-v` or `-latest`.
filename	The name of file in the corresponding project directory. Separate multiple filenames with white space. Use –a when you want all the files in the project.

Usage

Before using `Checkout`, change to the directory in your workspace to which the corresponding files will be checked out; the directory corresponding to the project hierarchy where the file resides. For example, to check out the files from the Tokens project, change to the corresponding Tokens directory:

```
cd $TaligentRoot/Taligent/Portable/OES/Tokens
```

Messages

After `Checkout` retrieves a file, it displays a message indicating the file's status and the version that was checked out.

```
% Checkout file.C simple.C hello.C
readonly file.C,1.5
readonly simple.C,1.9
readonly hello.C,1.5
```

`Checkout` *messages*

Locked revisions

Occasionally when you attempt to check out a file, `Checkout` tells you that someone else has the file checked out for modification. For example, if Arn has file.C is checked out, `Checkout` responds:

```
co error: revision 1.8 already locked by "arn"
```

There are several things you can do when you get this message.

Ask the other user to either check in or cancel the check out of the file. This is the safest procedure. To cancel a file checked out for modification:

```
Checkout -c file.C
```

Check out another version of the file on a branch. You can use `Checkout` to check out the file readonly with the intent of checking it back in on a branch. See "Check in files and designate the newly checked-in versions with the symbolic name ap_latest, even if another version of the file is already designated with that name (see "NameVersions" on page 28 for information about symbolic names):" on page 9 for specific information.

Change the access permission of the file. The `mro` script (modify-read-only) changes the access permission for you. To avoid conflict with your coworkers, use this sparingly. It is easy to forget that you changed the file access. The following week you might wonder why you are the only one in the group who can build your project (or the only one who can run anything).

```
mro File.C
```

`ListVersions` can help track down some of these problems. See "Latest" on page 26 for more information.

Examples

Check out the latest versions of file.C to the workspace for reading only:

```
Checkout file.C
```

Check out version 1.8 of each file:

```
Checkout -v 1.8 file.C hello.C simple.C
```

Check out all (–a) the files in the project directory out into the workspace directory. If a version of a file is designated as Master.55, Change.187, or Change.189, then check out the last version specified on the command line (`Checkout` handles this internally as if separate commands were issued):

```
Checkout -a -v Master.55 -v Change.187 -v Change.189
```

Get all sources for the *D34.FINAL* build. Check out all the files in the project directory with a symbolic name, and recurs to all subprojects in the project.

Note that `SyncSources` *does the same thing, but is quicker*

```
Checkout -v D34.FINAL -a -r
```

COMPAREVERSIONS

`CompareVersions` displays the differences between files in your workspace and files in the project. It can also compare two sets of files designated with symbolic names in the project against each other.

Syntax

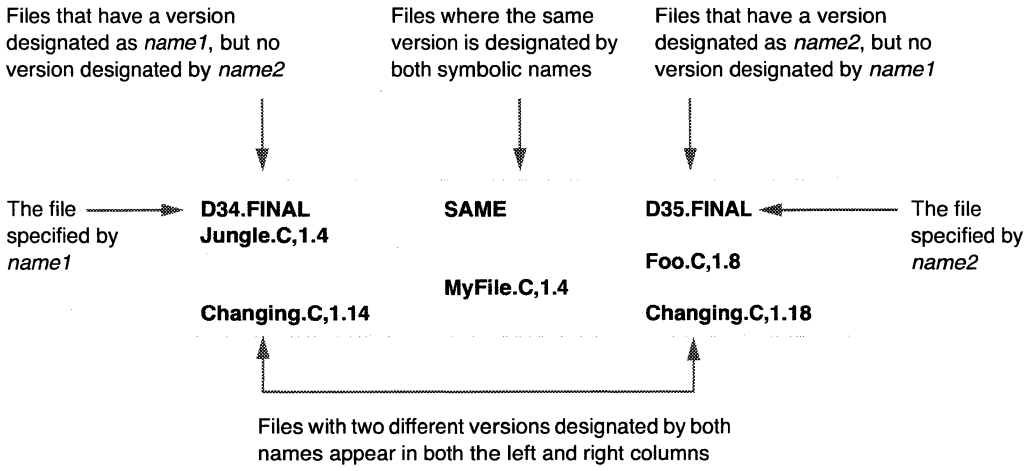
```
CompareVersions [-h] [-nnn] [-latest | name1 [ name2 ] ]
```

Arguments

–h	*headings:* suppress the column headings.
–latest	*latest:* Compare the files in your workspace against the latest versions of those files. Latest means *highest-numbered on the trunk*. For this option, the left column of the report contains the files in your space, and the right contains the latest versions in the project.
–nnn	The columns to include; the default is –123 for all three columns. For example, –1 outputs only column one, –2 only column two, and –23 outputs both columns two and three. Omitting a column suppresses all characters for that column—no spaces, no tabs.
name1 name2	Compare the files and versions designated by *name1* against those designated as *name2*, and report the similarities and differences. Omit *name2* to compare against the current files in your workspace. Either name can be a `NameVersions` symbolic name. See "NameVersions" on page 28.

Usage

`CompareVersions` prints a report in a three-column format.In the following example, version 1.4 of Jungle.C is designated with the name D34.FINAL. No version of Jungle.C is designated D35.FINAL; while the opposite is true of Foo.C version 1.8. Both names designate the same version of MyFile.C version 1.4. Changing.C has one version designated D34.FINAL, and a different version designated D35.FINAL.



Examples

Compare the files and versions designated with *D34.FINAL* against those designated with *D35.FINAL*, and report the similarities and differences.

```
CompareVersions D34.FINAL D35.FINAL
```

When you provide one name only, `CompareVersions` compares that name against the files in your workspace. In this case, the right column is labeled *(current)*.

```
CompareVersions D34.FINAL
```

Compare the files in your workspace against the latest versions of those files—the *highest-numbered on the trunk.*

```
CompareVersions -latest
```

LATEST

`Latest` reports the latest trunk version of files in the project directory. Unlike `ListVersions`, which reports the version of the files in your directory, `Latest` reports on the files in the repository.

Syntax

```
Latest [-r] [filename…]
```

Arguments

–r	*recursive:* operate recursively through workspace directories, skipping project directories that are not in the workspace.
filename	The name of a file in the corresponding project directory; you can specify more than one. Omit *filename* to report the latest version of all files in the current workspace directory.

Examples

Report the latest trunk workspace version of all files in the directory:

```
Latest
```

Report the latest trunk workspace version of each named file.

```
Latest file1.C file3.h
```

LISTVERSIONS

`ListVersions` reports the workspace version of each file in the current directory. It also tells you which files are checked out for modification.

Syntax

```
ListVersions [–c] [–m] [–n] [–x] [+c] [+m] [+n] [+x] [–r] [filename…]
```

Dash (–) options combine to suppress listing of multiple categories. Plus (+) options combine to list multiple categories.

Arguments

–c	Omit files checked out read-only.
+c	Only list files checked out read-only.
–m	Omit files checked out for modification.
+m	Only list files checked out for modification.
–n	Omit files not in the project.
+n	Only list files not in the project.
–r	*Recursive:* operate recursively through workspace directories, skipping project directories that are not in the workspace.

–x	Omit modify-read-only (mro) files.
+x	Only list modify-read-only (mro) files.
filename	The name of file in the corresponding project directory; you can specify more than one. Omit *filename* to get the versions of all files in the current (workspace) directory.

Usage

`ListVersions` looks for the `$Revision$` tag-line at the beginning of each file in your working directory (`SCMInsertHeader` inserts this line). If `ListVersions` cannot find the line, it prints a warning and attempts to figure out what the version is.

The report contains the filename, the version number of the file in the workspace, and one trailing mark:

(blank)	File is checked out for reading only
+	File is checked out for modification by you
#	File is MRO: workspace version is writable, but you do not have that version checked out for modification
<not in project>	File is not in the project (there is no corresponding file in the repository)

Binary files have a question mark (?) instead of a version number because the version number cannot be known; binary files do not contain the `$Revision$` tag-line.

Examples

To report the version of file.C in your current working directory:

```
% ListVersions file.C
file.C,1.1
```

`ListVersions` *message*

To report all files in the current directory, omit the *filename*:

```
% ListVersions
file1.C,1.1
file2.C,1.2+
file3.h,1.2<not in project>
```

`ListVersions` *messages*

Report the workspace version of all files except those not in the project, or those that are modify read only:

```
ListVersions -n -x
```

Report the workspace version of all files in the directory that are checked out for modification, or that are modify-read-only:

```
ListVersions +m +c
```

NAMEVERSIONS

`NameVersions` associates a symbolic name with a set of files in an SCM project or project hierarchy. It can designate a name, report what versions are designated with a name, display all names that designate any version of a file, and delete names. You can use the symbolic name when checking files out.

Syntax

```
NameVersions [-c | -C | -l | -L | -v version | -V version | -f | -d]
                [-r] symbolicName [filename...]
```

Arguments

–c	*current:* designate the current version(s) of the file(s) in your workspace with *symbolicName.*
–C	*current*: same as –c, but override an earlier definition of the symbolic name.
–d	*delete:* remove *symbolicName* so it does not designate any version of the files. If you omit *filename*, the symbolic name is completely deleted so it does not designate any version of any file.
–f	*find*: find versions designated with *symbolicName*. Display the version number of the files that the name designates. Omit *symbolicName* to display all the names that designate any version of any file in the current directory.
–l	*latest:* designate the latest version(s) of the file(s) in the project with *symbolicName*. This option does not look at files in your current directory, it only refers to the corresponding SCM directory. Note, this option can cause problems because latest means the *highest-numbered on the trunk*, not the most-recently checked in. Avoid this option.
–L	*latest:* same as `-l`, but override an earlier definition of the symbolic name.
–r	*recursive:* operate recursively through workspace directories, skipping project directories that are not in the workspace.
–v *version*	*version:* designate *symbolicName* to the *version* of the files. *version* can itself be a *symbolicName.*
–V *version*	*version:* same as –V, but override an earlier definition of the symbolic name.
filename	The name of file in the corresponding project directory; you can specify more than one. Omit *filename* to specify all files in the current workspace directory.

The lower-case options (–c,–l, and –v) let you designate a version with a name if that name does not currently designate any versions of the files, but these options do not let you change what version an existing name designates. The upper-case options let you change an existing name.

In the lower-case forms, if *symbolicName* already designates any version of any of the files you are applying it to, `NameVersions` stops and reports an error, and no changes occur. If applying the name to one file is not allowed, then it is not applied to any of them.

Usage

`NameVersions` names versions of files so that you can refer to the named set of files for checking in or out. For example, if each engineer designates their project *ReadyToBuild* when they are done, one person can later check out the entire project by that name and perform a complete build.

When you change what version of a file (or set of files) a symbolic name designates, that does not affect any other files which may have versions designated with that name. For example, Winner.C version 1.2 and BigWin.C version 1.5 are both designated with the name ReadyToBuild. If you run

```
NameVersions -V 1.7 ReadyToBuild BigWin.C
```

the result is that the name ReadyToBuild is moved to version 1.7 of BigWin.C. However, this does not affect which version of Winner.C, or any other file, is designated with that name.

Modes of operation

`NameVersions` has five basic modes of operation:

- Designate the current versions in the current directory (–c) with a name
- Find a name (–f)
- Designate the latest version (–l) of a file or files with a name
- Designate a particular version (–v) of a file or files with a name
- Delete a name (–d)

Binary files

NameVersions does not designate any version of a binary file when you use –c or –C. By definition, the SCM tools cannot tell what version of a binary file is in your workspace. However, if you include –b in addition to –c or –C, NameVersions designates the *latest* version of any binary files with *symbolicName*. For example, if you have the source file MunchData.C and a binary file TheData in a directory and you issue

```
NameVersions -c BuildVers MunchData.C TheData
```

you will get a warning telling you that NameVersions could not tell what version of TheData is in your current directory, and no version was tagged with the name. If you issue

```
NameVersions -b -c BuildVers MunchData.C TheData
```

then the current version of MunchData.C will be designated, and so will the *latest* version of TheData. In this manner you are telling the tool what version you have. You are specifying that you have the latest version, so that's the one you want designated with the name.

In this context, the *latest* version is the highest-numbered version on the trunk (no branches). This is the same version reported by `Latest`.

Examples

To name only two files:

```
NameVersions –c Defiant.1042 File1.C File2.h
```

Avoid the common mistake of inadvertently omitting the symbolic name when you specify multiple filenames.

To associate *Defiant.1042* with the latest versions of all the files in the project:

```
NameVersions –l Defiant.1042
```

To associate *Defiant.1042* with the with the 1.4 version of File1.C:

```
NameVersions –v 1.4 Defiant.1042 File1.C
```

To recursively find versions designated *Defiant.1042* in all files, from the current project directory down, and designate those versions of those files *D37.1*:

```
NameVersions -r -v Defiant.1042 D37.1
```

To designate to version 1.4 of file3.C with the name *Defiant.1042*, even if that name already designates another version:

```
NameVersions -V 1.4 Defiant.1042 file3.c
```

To list the files associated with *Defiant.1042*:

```
NameVersions –f Defiant.1042
```

If you no longer need a symbolic name, remove it with –d:

```
NameVersions -d Defiant.1042
```

To delete a particular file from a name:

```
NameVersions -d Defiant.1042 file3.c
```

NATIVEROOT

NativeRoot initializes your environment for Taligent Operating System (native) by setting the value of $TaligentRoot and several related shell variables. $TaligentRoot is the directory from which all your Taligent source directories and files descend. To initialize your environment for the layer environment, use SetRoot (see page 38).

CAUTION Never set $TaligentRoot directly—instead, always use NativeRoot because it also sets other important related variables.

Syntax

```
NativeRoot [-c] [-l] directoryName
```

Arguments

-c	*Create:* Make all the directories needed for installation.
-l	*Latest:* Use the latest directory structure available.
directoryName	The directory from which all your Taligent files will descend

Examples

```
NativeRoot ~/Work
NativeRoot -O ~/Work
```

SCMADMIN

SCMAdmin reports and sets the attributes of a file, and breaks another user's lock on a file.

Syntax

```
SCMAdmin [-v Version] [ rcsOptions ] filename
```

Arguments

rcsOptions	Options to pass to rcs.
-v *Version*	Administer this version of the file; can be a number or symbolic name.
filename	The name of file in the workspace directory

Examples

Unlock the version of the file designated *D34.FINAL*; –u is the unlock option to break the lock on file.C. (Be sure to notify the owner of the lock first.)

```
SCMAdmin -u -v D34.FINAL file.C
```

Mark file.C as *binary*:

```
SCMAdmin -ko file.C
```

Mark file.C as a normal text (non-binary) file:

```
SCMAdmin -kkv file.C
```

SCMCREATEDIRECTORIES

`SCMCreateDirectories` examines the project and creates corresponding directories in your workspace.

Syntax

```
SCMCreateDirectories [-d]
```

Arguments

–d	Refrain from creating directories named "Docs" (note the initial capital). These are the class and member description files directories. Using this option you can create a workspace which does *not* include those directories. This option is for use by Pre Build and Integration teams (PBIs) and the build room, because having those directories slows down their check out operations.

Usage

`SCMCreateDirectories` creates directories recursively starting with your current directory. That is, if you start it from `$TaligentRoot/Taligent/NetComm`, it creates directories that exist in the project below `$TaligentSCMRoot/Taligent/NetComm`.

SCMDIFF

`SCMDiff` uses `diff` to compare a file in your workspace against a version of that file in the project.

Syntax

```
SCMDiff [ -v version [ -v version2] ] [ diffOptions ] filename
```

Arguments

diffOptions	Options to pass to `diff`.
-v *Version*	Compare the workspace file against *Version*. *Version* can be a version number or a symbolic name.
-v Version1 -v Version2	Compare these two versions of the file.
filename	The name of file in the corresponding project directory. If you omit –v options, compare the current working file against the locked version of that file, or against the latest version on the trunk if you do not have it locked.

Examples

Compare version 1.3 of file.C against the version designated with *D34.FINAL*:

```
SCMDiff -v 1.3 -v D34.FINAL file.C
```

Compare file.C in your workspace against the latest version on the trunk. Pass -c to `diff` to supply context around the differences:

```
SCMDiff -c file.C
```

Compare file.C in your workspace against the version designated with *D34.FINAL*. Pass -b to `diff` to ignore differences in indenting and spaces:

```
SCMDiff -b -v LastDRelease file.C
```

SCMFETCH

SCMFetch is used by xcdb to get a copy of a source file for debugging when xcdb can't find a copy on its own.

Syntax

```
SCMFetch [-build] [-xfile fname] [-xpath dirname] [-which] [filename...]
```

Arguments

-build	Update the SCMFetch cache.
-which	Report how SCMFetch found the file you asked for, rather than actually producing its contents on standard output.
-xfile *fname*	Exclude the file *fname* when updating the cache
-xpath *dirname*	Exclude the directory *dirname*, and its children, when updating the cache.
filename	Look for *filename* in the source code hierarchy. If found, echo its character content to standard output (console).

Usage

SCMFetch maintains a cache of the files in the project; the cache contains the containing directory name of each file in the project. It maintains this cache in the root directory of the SCM repository, $TaligentSCMRoot.

When the debugger calls for a file, SCMFetch searches for it in this order:

- If $TaligentSCMFetchPath is set, SCMFetch looks in those directories for the file. You can specify more than one directory by separating them with colons: /home/joe/dir1:/home/joe/dir2
- If $TaligentSCMFetchPath is not set, or if the file is not in those directories, SCMFetch consults its cache for the project directory that contains the file. It then looks for the file in your corresponding workspace directory.
- If the file is not found in the workspace, SCMFetch checks the file out from the repository. If the file $TaligentRoot/TaligentSCM/BuildName exists, then SCMFetch uses that file's contents as a version name to check out. Otherwise it checks out the latest version on the trunk.

  The directory that the file belongs in must exist in your workspace. Make sure your workspace contains all directories with SCMCreateDirectories.
- If the file is not found in the repository, SCMFetch writes an error message to stderr and exits.

Sometimes there is more than one file in the world with the same name. To ensure SCMFetch finds the file that you want, set $TaligentSCMFetchPath to include the file's directory. This ensures that SCMFetch searches the directory before searching the cache.

When SCMFetch checks a file out from the repository, it uses the BuildName file to supply a -v version to Checkout. If you are working with a version that does not match the version in BuildName, manually check out the versions of the files that you are using so SCMFetch uses the ones in your workspace.

Building the cache

The -build option builds a cache file. When you build a cache, use –xfile and –xpath to prune the cache of unwanted directory and file entries, and to ensure that the correct file appears when two files in the repository have the same name.

Use –xfile *nnn* to omit from the cache those files whose names end with the pattern *nnn*. SCMFetch compares the pattern against the full path name of each file before adding the file to the cache. You can supply more than one -xfile.

Use -xpath *ppp* to omit from the cache those directories whose names end with the pattern *ppp*. SCMFetch compares the pattern against the full path name of each directory. If it matches, then SCMFetch omits that directory and all its subdirectories. You can supply more than one -xpath.

Example

Assume that you have directory Sources/Tools in your repository, and that directory has some files in it. To split Tools into DevelopmentTools and AnalysisTools, you cannot just delete the Tools directory because then you could not build versions from before the split. To keep SCMFetch from finding the files in Tools:

```
-xpath /Tools
```

Assume that Scripts is another directory containing files. To split it and put *some* of the files in the new directory ExtraScripts, you have to make sure that the old file do not appear to SCMFetch:

```
-xfile /Scripts/old1 -xfile /Scripts/old2
```

The patterns in the example start with slashes because they are matched against filenames like Sources/Scripts/old1 and Sources/ExtraScripts/old1. Without the leading slash, both of these would match and would both be omitted.

SCMInsertHeader

`SCMInsertHeader` prepares files for initial check in to SCM. It removes old SCCS (Source Code Control System) tag lines from previous sessions, and it inserts an SCM `$Revision:$` tag-line. It ignores files that already have a `$Revision:$` or `$Header:$` header-tag.

Syntax

```
SCMInsertHeader filename...
```

Arguments

filename	The name of file in the corresponding project directory; you can specify more than one.

Usage

If *filename* contains a SCCS tag lines, `SCMInsertHeader` uses the comment leader on those lines for the new SCM tag line. Otherwise, if there is no SCCS tag line, `SCMInsertHeader` guesses a comment based on the filename:

- `//` comments for .C, .c, .h, and .PinkMake files
- `#` comments for .Make files, and for files ending with *Makefile* and *makefile.*

If file does not contain a SCCS tag, and if `SCMInsertHeader` cannot determine which comment leader to use from *filename*, `SCMInsertHeader` does not modify the file and a prompt instructs you to add an SCM tag manually.

`SCMInsertHeader` modifies the file in place; it does not make a copy.

SCMLog

`SCMLog` displays the revision history of the file.

Syntax

```
SCMLog [-v version] [ rlogOptions ] filename
```

Arguments

rlogOptions	Options to pass to `rlog`.
-v *version*	Compare the workspace file against the specified version.
filename	The name of file in the workspace directory.

Usage

`SCMLog` takes the same arguments as `rlog`; see `man rlog` on-line for more details. Unlike `rlog`, however, you cannot pass a range of version numbers to `SCMLog`, only a single version number or name, and you must use –r to pass it a version number.

SCMNORMALIZE

`SCMNormalize` produces a *normalized* form of either a file or a directory name. The *normalized* form is a full pathname, not a relative pathname, and starts with `/mnt` if the *real* directory starts `/tmp_mnt/mnt`.

Syntax

```
SCMNormalize [filename | directoryName]
```

Arguments

directoryName	The name of a directory in your workspace.
filename	The name of the file in the workspace.

Usage

`SCMNormalize` prints the pathname to standard output.

Use `SCMNormalize` when setting the shell variable `$TaligentRoot`, which must be in normalized form in order for the SCM tools to work properly. `SetRoot` and `NewRootCommands` use `SCMNormalize` to set `$TaligentRoot` for you.

Examples

Here are two examples of `SCMNormalize`:

```
% SCMNormalize ~tsoi
/home/tsoi

% SCMNormalize /usr/taligent/bin
/usr/talilocal/taligent/bin
```

`SCMNormalize` *result*

`SCMNormalize` *result*

SCMPROJECTFILE

`SCMProjectFile` reports the full path name in the `$TaligentSCMRoot` repository of a file in your working directory. It can also report the path to the corresponding working directory's path.

Syntax

```
SCMProjectFile [filename]
```

Arguments

filename	The name of the file in the workspace directory. This file does not have to exist in your workspace. If you omit *filename*, it returns the full pathname of the corresponding project directory.

Usage

Use `SCMProjectFile` when writing scripts that need pathnames. `SCMProjectFile` prints the pathname to standard output.

If the current directory does not descend from `$TaligentRoot`, `SCMProjectFile` reports an error and exits with nonzero status.

SETROOT

SetRoot initializes your Taligent Application Environment environment by setting the value of $TaligentRoot and several related shell variables. $TaligentRoot is the directory from which all your Taligent source directories and files descend. To initialize your environment for the layer environment, use NativeRoot (see page 31).

CAUTION Never set $TaligentRoot directly—use SetRoot instead because it also sets other important related variables.

Syntax

```
SetRoot [-O] directoryName
```

Arguments

-O	*Optimize:* turn optimization on for your compiles (affects the setting of $CompileOptions)
directoryName	The directory from which all your Taligent files will descend

Examples

```
SetRoot ~/Work

SetRoot -O ~/Work
```

SYNCSOURCES

SyncSources compares the files in the workspace against those versions of the files in the project designated with a specified symbolic name, and checks out the files necessary to get your workspace in sync with the project.

Syntax

```
SyncSources [-a] [-e] [-r] [-latest] [-s] [-d] [-w]
                { syncName | -v syncName [-v syncName]… }
```

Arguments

-a	*all:* report on files that are the same and therefore are not checked out, and on files in the project that do not have a version named *syncName.*
-d	*delete:* delete files in the workspace that are not in the repository (not controlled). It executes rm commands unless you also include -s, then it prints the rm commands to standard output. It does not generate rm commands for directories.
-e	*exhaustive:* report files that are in your workspace and which are not in the project (a condition necessary for a *clean build*). –e also retrieves the same files as –a.
–latest	Designates the *latest* versions of files, not just those which are designated by a symbolic name. In this context, the *latest* version is the highest-numbered version on the trunk (no branches). This is the version that Latest reports.

-r	*recursive:* operate recursively through subdirectories in your workspace, silently skipping subdirectories in your workspace that aren't in the project, and reporting subdirectories that are in the project and not in your workspace.
-s	*script:* generate a script on standard output which, if executed, would perform the check outs.
–v *syncName*	The symbolic name that designates the versions of the files to compare against the files in your workspace.If you include more than one *syncName*, `SyncSources` behaves as if you gave multiple commands, one for each symbolic name, in the order given.
–w	Displays which symbolic name caused a version of a file to be checked out.

Usage

`SyncSources` checks out all files with a version designated with *syncName*, similar to `Checkout -r -v syncName`, but skips the files for which you already have the right version. In this case, it is an *optimized* check out.

`SyncSources` does not overwrite files you have checked out for modification, or modify-read-only files; for those it reports an error.

`SyncSources` reports the reason for each check out that it does, such as whether the file was missing in your workspace, if was there already, what version was there, and what version is being checked out.

Examples

Check out and sync all files designated Master.55:

```
SyncSources Master.55
```

The –v option is required for multiple versions, like this:

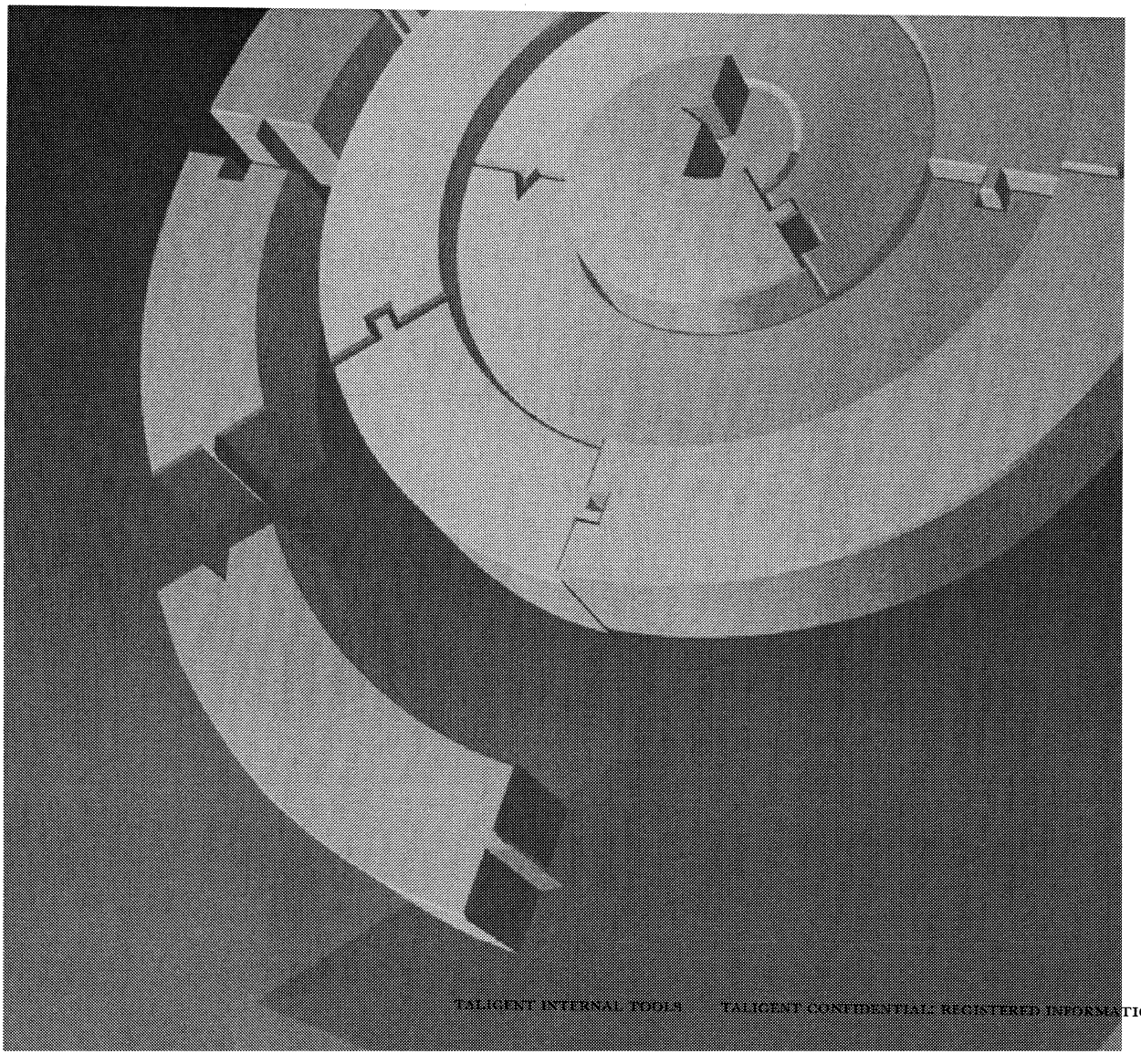
```
SyncSources -v Master.55 -v Change.187
```

Checks out and synchronizes all (–a) the files in the project directory out into the workspace directory. If a version of a file is designated Master.55, Change.187, or Change.189, then check out the last one specified on the command line (`SyncSources` handles this internally as if separate commands were issued):

```
SyncSources -a -v Master.55 -v Change.187 -v Change.189
```

Check out the latest versions of the files in the workspace directory:

```
SyncSources -latest
```



CHAPTER 4

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.

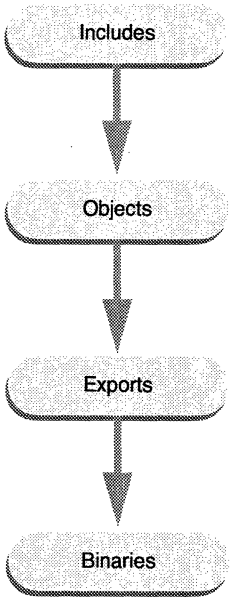
TALIGENT BUILD TERMINOLOGY

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 43 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 the layer or layer applications, use `MakeSharedLib` (see page 69 for more details). When building native, the link is handled automatically by `Universal.Make.Intel`, which calls Plink.
- *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 68 for more details.)

THE BUILD PROCESS

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



Exports all public header files for clients

Compiles all .C files into .o files

Combines all .o, and generates .e files for clients

Generates all shared libraries and executables

The build process makes header files (*.h) by copying them from the project into a common directory where other projects (clients) can access them

The build process makes export files (*.e) by compiling *.C files, combining them into one .o (or .a), and then using `MakeExportList` to generate the .e file (see "MakeExportList" on page 65)

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.

NOTE Chapter 6, "CreateMake," describes makefiles in detail.

Makefile description syntax

```
TypeOfTarget TargetName {
    Label:
        FileList
    Label:
        FileList
}
```

Type of target, common types include Library, Program, ParentObject, *and* SubProjectList

Name of the target

Identifies the build topic, typically Source, Link, *or* PublicHeaders

The files to process

Target types

`CreateMake` generates different build rules for each type of target. Here are a few common target descriptions; for more, see Chapter 6, "CreateMake."

Library

Generates rules to build a shared library.

```
Library WidgetLib {
    Source:
        AbstractWidget.C
        Widget.C
    PublicHeaders:
        Widget.h
    Link:
        TestFrameworkLib
        ToolboxLib
}
```

Build WidgetLib, also generates Widget.e to allow other Widget.h files to link in.

WidgetLib is built from these two files

Export Widget.h to allow other projects to use Widget objects

Specifically link with these files

Program

Generates rules to build an application.

```
Program ShowWidget {
    Source:
        ShowWidget.C
}
```

Use all system libraries because there is no Link label

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

Exported for clients

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

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 your 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 66.

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 49.

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`). For more information, see "CreateMake" on page 60.

Universal.Make

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

NOTE Each target platform has a separate version of `Universal.Make`. For native builds, the file is `Universal.Make.Intel`.

`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.

NOTE Before you build anything with `Makeit`, follow the installation procedures in Chapter 2, "Working in the AIX environment" to check out a correct version of `Universal.Make` or `Universal.Make.Intel` into your `$TaligentRoot/Taligent` directory.

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

The {} bound variable references in shell scripts.

```
setenv TaligentIncludes ${TaligentRoot}/PinkIncludes
setenv TaligentExports ${TaligentRoot}/Exports
```

Variable	Path to
LIBPATH	Taligent shared libraries used during runtime.
TaligentBATRoot	Root of the area where all BAT scripts, data, and results reside. BAT libraries, and servers go in the nontest (standard library, server, program) areas.
TaligentBinaries	Taligent runtime binaries.
TaligentDefaultHomePlace	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.
TaligentExports	Taligent shared library interface files that developers link with to access Taligent shared libraries.
TaligentExtensionIncludes	Directory containing interfaces to system extension developers.
TaligentIncludes	Main `#includes` directories used in Taligent builds.
TaligentIncludesDir	Base parent directory of all Taligent `#includes` (this is the parent of $TaligentIncludes, $TaligentExtensionIncludes, and $TaligentObsoleteIncludes).
TaligentLibs	Directory for certain nonshared libraries.
TaligentObsoleteIncludes	Directory containing interfaces that should not exist in the SDK release but cannot, or that have not had their dependencies successfully removed. This directory will go away by SDK2.
TaligentPlacesRoot	Repository where Places for the machine reside.
TaligentPrivateIncludes	Private `#includes` used internally.
TaligentRoot	The base of everything in the build and runtime system.
TaligentSCMRoot	The repository for Pink source.
TaligentSharedLibs	Taligent runtime shared libraries.

Variable	Path to
TaligentSource	The root of the source tree. `TaligentSource` is not used by the build, but is used by some tools that the Build room uses. TaligentSource is the root of the native and layer source tree; eventually there will be settings for Hoops, CompTech, and possibly more.
TaligentSystemDataRoot	Repository for system data files. These are typically configuration files, not first class user data such as movies, images, or sounds.
TaligentSourceRoot	Root of the Taligent source tree hierarchy.
TaligentSystemLibraries	Repository for system software shared libraries.
TaligentSystemPrograms	Repository for system software shared libraries.
TaligentSystemRoot	Root of the Taligent system software area.
TaligentSystemServers	Repository for system software servers.
TaligentTemporaries	Repository for temporary files until people use real Pluto temporary file support.
TaligentTestLibraries	Repository for system test shared libraries.
TaligentTestPrograms	Repository for system software shared libraries.
TaligentTestServers	Repository for the test servers.
TaligentTools	Internal tools.
TaligentToolsEtc	Additional internal tools, scripts, etc.
TaligentUniversalMake	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	`xlC` 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	`xlC` 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
```

SetRoot and NativeRoot

`SetRoot` defines the standard values for all the environment variables that the Taligent build environment requires. You can review the complete `SetRoot` list of the environment variables (and descriptions) by looking at the `/usr/taligent/bin/NewRootCommand` script. Always use `SetRoot` to initially set the variables and pathnames. If you need to change a variable, do so after running `SetRoot`.

NOTE `SetRoot` is the layer command. If you are working on Taligent Operating System, use `NativeRoot` instead. For more information, see "Initialize your environment" on page 5 and "Prepare your environment" on page 6.

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.

NOTE Before you begin this section, make sure your initial set up is correct (see Chapter 2, "Working in the AIX environment") including checking out the `Universal.Make` file.

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 "Emacs" on page 162.

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
```

NOTE When compiling for Taligent Operating System, `NativeRoot` automatically sets up your environment so that `Makeit` uses the `-intel` argument to generate `Universal.Make.Intel` instead of `Universal.Make`.

The build log

What follows is the build log; yours should look similar.

Makeit messages

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.

```
1  ### Makeit:  No makefile found in `/home/EeeDee/SimpleSample'.
2  ###             However one will be built from `SimpleSample.PinkMake'.
3  # CreateMake > SimpleSample.Make;
```

The Includes phase

Since SimpleSample.PinkMake did not specify any public header files, `Makeit` did not build any include files.

```
4  #
5  # Making "Includes" for "/home/EeeDee/SimpleSample"...
6  # make -f SimpleSample.Make Includes
7  #
8  make: Nothing to be done for 'Includes'.
```

The Objects phase

Compiles hello.C to hello.o, and contains the `make` line that `Makeit` called.

```
9  #
10 # Making "Objects" for "/home/EeeDee/SimpleSample"...
11 # make -f SimpleSample.Make Objects
12 #
13 # Compile hello.C to produce hello.o
```

The Exports phase

Did not build a shared library because SimpleSample did not build an export file.

```
14 #
15 # Making "Exports" for "/home/EeeDee/SimpleSample"...
16 # make -f SimpleSample.Make Exports
17 #
18 make: `Exports' is up to date.
```

The Binaries Phase

Creates the executable application by calling `MakeSharedApp` (as echoed from `make`). For more information, see "MakeSharedApp" on page 41

```
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/Expo
   rts/RuntimeLib.e /home/EeeDee/work/Exports/OpixLib.e /home/EeeDee/work/Exports/T
   oolboxLib.e /home/EeeDee/work/Exports/TimeLib.e /home/EeeDee/work/Exports/TestFr
   ameworkLib.e /home/EeeDee/work/Exports/HighLevelAlbert.e  /home/EeeDee/work/Expo
   rts/LowLevelAlbert.e /home/EeeDee/work/Exports/AlbertPixelBuffers.e
```

The Copy phase

Copies the built application to `$TaligentBinaries`, the standard location for executable files, and leaves a copy in the current directory.

```
24 SmartCopy SimpleSample /home/EeeDee/work/TaligentBinaries
```

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
Hi there everybody!
%
```

The Taligent AIX Layer prints a time-stamp when it runs an application.

NOTE See "Starting and stopping Taligent Operating System programs" on page 13 for information about running the Simple Sample program on Taligent Operating System.

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/Expo
rts/RuntimeLib.e /home/EeeDee/work/Exports/OpixLib.e /home/EeeDee/work/Exports/T
oolboxLib.e /home/EeeDee/work/Exports/TimeLib.e /home/EeeDee/work/Exports/TestFr
ameworkLib.e /home/EeeDee/work/Exports/HighLevelAlbert.e  /home/EeeDee/work/Expo
rts/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.

compileoption: -D__DEBUG__ -DUSE_FILE_SEGS

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

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

localheaderdir: $(TestHeaderDir)

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}
}
```

Used by all compile commands.

Copy these `make` *commands into the beginning of the generated makefile.*

Directory of headers to export.

Dependencies and makefile commands for creating the runtime library.

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 JustAViewLib {
source:
    MyView.C

link:
    {SimpleLinkList}        // Minimal set
}

binary JustAView {
source:
    Main.C

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

Add link targets

Add link targets

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 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, 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:
HighLevelAlbert
```

Link tag to add

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

Add link targets

```
library JustAViewLib {
source:
    MyView.C

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

binary JustAView {
source:
    Main.C

link:
    {SimpleLinkList}
}
```

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}
}
```

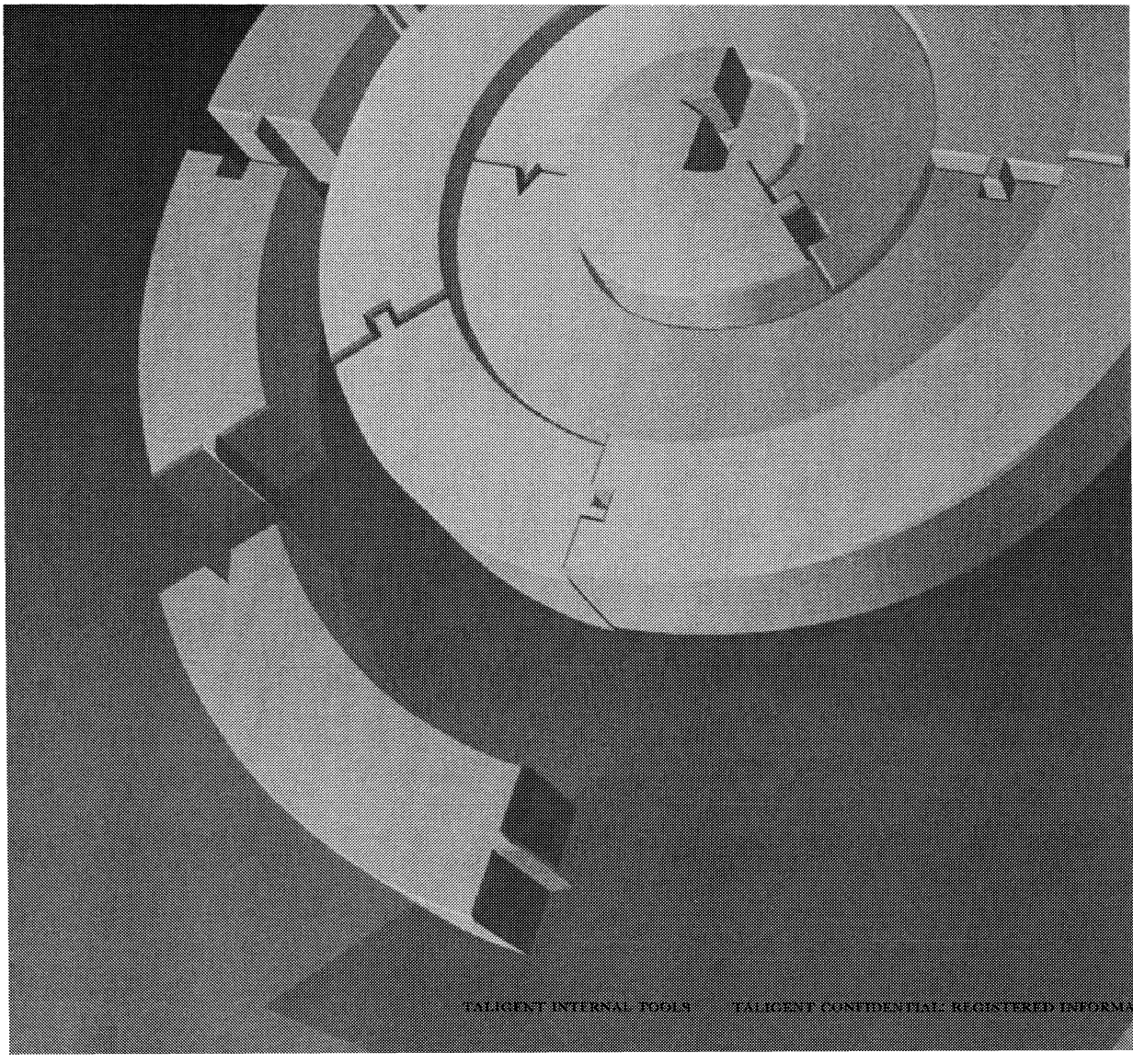
Add link targets

Repeat the process until `Makeit` completes the build.

SYSTEM BUILDS

Often you need to build the entire system to ensure that your application pieces are functioning together. You can install a copy of the latest build to your file system with `InterimInstall` (layer) or `NativeInstall` (native). These scripts copy a set of headers, libraries, and shared libraries to your local system. Once you install a build and its associated files, you can modify, debug, or build on top of that particular build.

For more information about `InterimInstall`, see "InterimInstall" on page 64, and for `NativeInstall`, see "NativeInstall" on page 70. To install a system build, follow the instructions provided in "Install the build" on page 5 or "Install the Native TalOS build" on page 7.



CHAPTER 5

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]… [-target target] [-I includePath]…
           [-noum] [-vers] > outputfile
```

Arguments

–D *define*	Include the specified definition during processing.
-fast	Preprocess the source files and create a single .c that `#includes` the source files to build each target. this results in faster builds, but is *not* to be used for final builds.
–target *target*	Generate a makefile for a specific target. Currently used only by Taligent Operating System and the *target* is `intel`. Use `Universal.Make.Intel` instead of `Universal.Make`.
–I *includePath*	Add the path to the `#include` directory search-list.
–noum	Generate a makefile that does not rely on Universal.Make for processing.
outputfile	The file containing the new makefile. If you omit *outputfile*, output goes to `stdout`.
sourcefile	The input file to process is usually a *.PinkMake filename. If you omit *sourcefile*, `CreateMake` assumes the current directory name is the project. For example, if the current directory is `/TestLib`, the *sourcefile* is TestLib.PinkMake.
-vers	Echo the current version and copyright information to `stderr`. 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 66). `Makeit` executes `CreateMake` if the makefile is out-of-date or missing.

See Chapter 4, "Makefiles," for a formal definition and syntax for the makefile descriptions.

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 `*.PinkMake` 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 `*.PinkMake`'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.
   - B. 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.

INTERIMINSTALL

`InterimInstall` is a script that automates the installation process for the layer; to install native, use `NativeInstall`. `InterimInstall` installs the most recent build by default, or can install a specific build.

Syntax

```
InterimInstall [ -l  |  [ -S | -D | -O ] [-b] [-r releaseName ]  ]
```

Arguments

–D	Install the Debug release.
-b	*Blast* the release currently installed on your system. Most of the files in the Taligent directories are not writable, but must be removed before a new build can be installed over them. This option removes the pertinent directories under $TaligentRoot, but does not modify $TaligentRoot/Taligent except to remove universal.Make.
–l	List the builds currently available for downloading. You cannot use this with any other option.
–O	Install the optimized release.
-r *releaseName*	A specific release to install. If you do not specify a release, `InterimInstall` downloads the current build.
-S	Install the stripped release.

Example

To install MS-0.07 debug and remove the existing release:

```
InterimInstall -D -b -r MS-0.07
```

→B blast more

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.

`IPCPurge` is used within the layer only; the native environment doesn't have an equivalent function.

MAKEEXPORTLIST

`MakeExportList` generates an .e file from an .o file (which is a combination of one or more `xlC` 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

–c	Do not build subprojects. By default, `Makeit` operates recursively on subprojects from the bottom up, executing targets at every project it finds in a *subproject{}* block.
–D	Do not rebuild a make file, even if it is out of date.
–i	Do not stop when errors are encountered. This is passed on to `make` as `-i`.
–fast	`CreateMake` option; Makeit passes this option to `CreateMake`.
–M	Force all makefiles to be rebuilt on the fly by calling `CreateMake` even if files are up-to-date.
–T	Traverse the project tree, but do not build anything.
VAR=*value*	Assign *value* to the variable named *VAR*. `Makeit` passes this expression to `make` to alter makefile variable usage.
–vers	Echo the current version and copyright information to `stderr`.
Targets	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 `complete` to build the four targets. `Makefiles` is a special *target* that generates a new makefile, but does not build anything. Use this for debugging. `Makeit Makefiles`

Usage

Go through each build process target (Includes, Objects, Exports, and Binaries) and build the necessary dependencies.

```
Makeit
```

To build DemoApp, and its subprojects:

```
Makeit DemoApp
```

A common mistake is to tell `Makeit` to build one target (like the previous example), and not realize that it will execute `make DemoApp` on all subprojects—many of which do not have a target DemoApp. To prevent `Makeit` from building subprojects:

```
Makeit -c DemoApp
```

To require `Makeit` to execute only the Includes and Exports targets in each directory.

```
Makeit Includes Exports
```

Passing options to make

`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 (`-i` option for `make`):

```
Makeit -i Objects
```

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:

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

Creating makefiles

`Makeit` can build makefiles on the fly. `Makeit` rebuilds a makefile if:

- the *.Make file does not exist
- the *.PinkMake file is newer than the *.Make file
- you specify `-M` to override the automatic makefile generation

`Makeit` uses `CreateMake` to translate the makefile descriptions (*.PinkMake) to makefiles (*.Make). `CreateMake` is akin to the `CreatePinkMakefile` tool used by the native system in MPW. For more information see "CreateMake" on page 60.

Universal.Make

To prevent duplication in each makefile, and to allow for more flexibility, `Makeit` includes Universal.Make in every makefile (*.Make). Universal.Make contains global targets and rules, such as Includes, Objects, Exports, and Binaries.

NOTE Be sure to follow the installation procedures in "Setting up for Taligent Application Environment" on page 3 and check out a correct version of Universal.Make to your `$TaligentRoot/Taligent`. Do this before you attempt to build anything with `Makeit`.

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 `xlC` command with special options). `MakeSharedApp` is a layer application; the native environment doesn't have an equivalent function because `Universal.Make.Intel` automatically calls `Plink` to handle this.

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. `MakeSharedLib` is a layer application; the native environment doesn't have an equivalent function because `Universal.Make.Intel` automatically calls `Plink` to handle this.

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 *file*	An additional file to register.
–c	Detects linking against `.e` files that don't have corresponding library files.
–e *pattern*	Excludes files matching the pattern.
–E *file*	Excludes the files listed.
–i *pattern*	Includes files matching the pattern.
–I *file*	Includes the files listed.
–t	Includes the test libraries. By default, they aren't included.
–v	Lists—to `stdout`—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
```

`mop` is used within the layer only; the native environment doesn't have an equivalent function.

NATIVEINSTALL

`NativeInstall` is a script that automates the installation process. `NativeInstall` installs the most recent build by default, or can install a specific build.

Syntax

```
NativeInstall [ -l  |  [ -D ] [-b] [-r releaseName ]  ]
```

Arguments

-b	*Blast* the release currently installed on your system. Most of the files in the Taligent directories aren't writable, but must be removed before a new build can be installed over them. This option removes the pertinent directories under $TaligentRoot, but does not modify $TaligentRoot/Taligent except to remove `Universal.Make`.
–l	List the builds currently available for downloading. You cannot use this with any other option.
-r *releaseName*	A specific release to install. If you do not specify a release, `NativeInstall` downloads the current build.
-T	Do not install tools into $TaligentRoot/ToolsDir. Instead, let `NativeInstall` install tools that are synchronized with your source code.

Example

To install N10.1 and remove the existing release:

```
NativeInstall -b -r N10.1
```

RP

`rp` loads and runs a Taligent Operating System program that was built with shared libraries; for programs that don't use shared libraries, use `runpink` instead.

Syntax

```
rp [ +a args ] programName
```

Arguments

+a *args*	Pass the specified arguments to the program.
programName	The shared-libraries built Taligent Operating System program.

NOTE `rp` is only available within Taligent Operating System. For layer programs, use `StartPink`.

Usage

Before invoking `rp`, you need to start up the Shared Library server. To do this:

```
rp _libserver &
```

Then, start up your program with:

```
rp MyProgram
```

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

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

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.

RUNPINK

`runpink` loads and runs a Taligent Operating System program that don't use shared libraries; for programs that use shared libraries, use `rp` instead.

Syntax

```
runpink [ +f ] [ +a args ] programName
```

Arguments

+a *args*	Pass the specified arguments to the program.
+f	Do not invoke the built-in runpink debugger. Just execute the program.
programName	The Taligent Operating System program.

NOTE `rp` is only available within Taligent Operating System. For layer programs, use `StartPink`.

Example

Start a program, and pass two arguments:

```
rp +a "couch" MyProgram
```

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 *sharedLib*	Create cache of symbols required for flatten/resurrect.
–da *sharedLib*	Create cache of all formal symbols (rarely used).
–r *sharedLib*	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`.

`SharedLibCache` is used within the layer only; the native environment doesn't have an equivalent function.

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`.

`slibclean` is used within the layer only; the native environment doesn't have an equivalent function.

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

destFile	The destination of the file being copied.
sourceFile	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 *applicationName*	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.
```

`StartPink` is used within the layer only; the native environment doesn't have an equivalent function.

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 70.

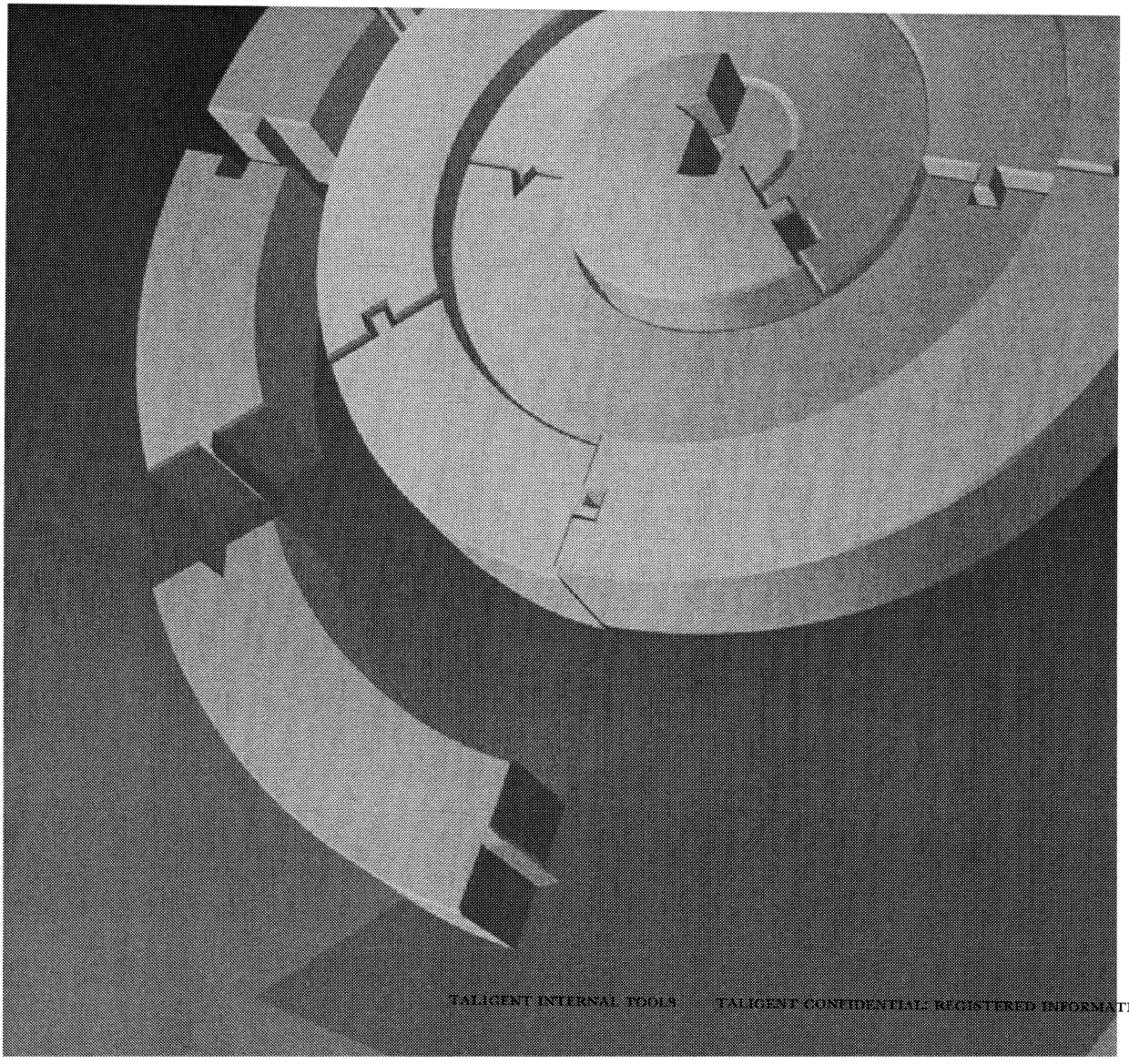
Syntax

```
StopPink
```

Usage

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

`StopPink` is used within the layer only; the native environment doesn't have an equivalent function.



CHAPTER 6

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 43.

NOTE When building for Taligent Application Environment, references to compile and link methods are referring to the IBM `xlC` compiler and linker. When building for Taligent Operating System, references to compile and link methods are referring to the Comptech-on-AIX C++ compiler and the Plink-on-AIX 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 `cplusoption` is a synonym for `compileoption` that will soon be eliminated. Change all occurrences of `cplusoption` 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

exportTags	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)/SharedLibaries:
testlibrary:
    SampleExportTestLibrary     // to $(OPD)/SharedLibraries:
            TestSharedLibaries:
testdata:                       // to $(TaligentTests)TestData:
    SampleExportTestData
testscript:                     // to $(TaligentTests)TestScripts:
    SampleExportTestScript
}
```

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: heapSizex
```

Argument

heapSize	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

name	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

linkFiles	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 79) 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. This feature is supported when building for Taligent Operating System with Comptech-on-AIX.

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 `LibIO.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:LibIO.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.

```
parentobjetdir: …/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, compiles 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 $TaligentIncludes.

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 $TaligentIncludes.

```
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 78 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 79 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 84 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 89 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 96 for more information.

Syntax

```
testserver name {
}
```

TOOL

This target is similar to the `binary` target. See "binary" on page 79 for more information.

Syntax

```
tool name {
}
```

TRIMDEPENDENCIES

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

Syntax

```
trimdependencies {
    headerPaths
}
```

Argument

headerPaths	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 $TaligentIncludes and $TaligentPrivateincludes

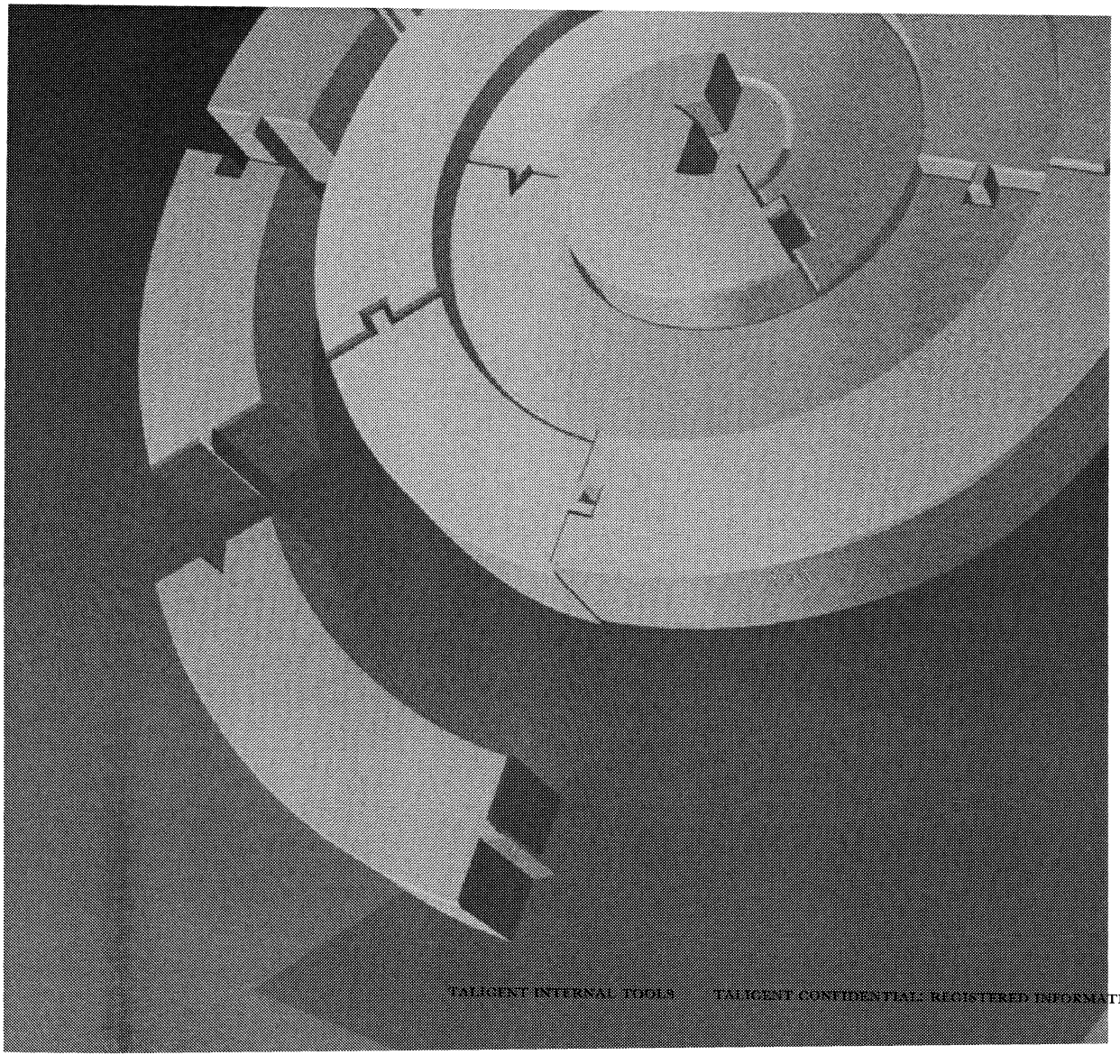
By default, `CreateMake` includes the list of dependent header files found in $TaligentIncludes and $TaligentPrivateincludes. In most cases, these headers do not change and the extra dependencies result in larger make files that take longer to process. With `trimdependencies`, `CreateMake` removes any 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 7

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 `0xDEAFBEED`.

  Deleted memory gets filled with the pattern `0xFEEDDEAD`.
- **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.

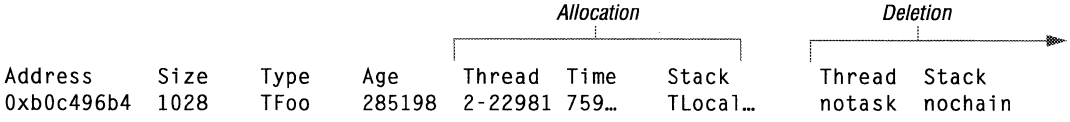
Stack crawl—the function that called `new()` or `delete()`. For old blocks, this field is empty. The stack crawl consists of several function names, separated by '|' characters. The first function name is the innermost. It was called by the next function name, and so forth. In the example, the stack 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 105 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 102 for more information.



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 analyzer has an extensible discipline architecture consisting of a set of instances of concrete subclasses of THeapDiscipliner. (These objects are equal, by default, if they are of the same type.) This class has the virtual method CheckBlockEvent; subclasses override this to provide discipline behavior.

The heap model has several varieties of discipline are built into it (there is no THeapDiscipliner subclass for these anomalies):

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 toll 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.

Garbage finding

Garbage finding is locating blocks in the heap that represent storage leaks. *Mark-and-sweep* garbage finding looks in the address space for pointers to blocks, and if there are no pointers to a block, considers the block garbage. This technique searches other blocks in the heap, local variables on the stack, and the static data areas. *Allocation-deletion matching* is a simpler scheme that considers a block garbage at some point in time if it has been allocated but not deleted. The latter scheme has fewer dependencies, and so it is more portable, but it gives more false positives.

Garbage finding is not implemented as a subclass of THeapDiscipliner. It is handled separately.

CLASS DESCRIPTIONS

The main classes are TLocalHeapMonitor, TLocalHeapAnalyzer, and THeapMirror. In addition, these classes pull in a few auxiliary classes.

Local heap tool

The classes TLocalHeapMonitor and TLocalHeapAnalyzer allow you to analyze heaps in the same team under programmatic control.

TLocalHeapMonitor

`TLocalHeapMonitor` begins monitoring of a TStandardMemoryHeap when TLocalHeapMonitor object is constructed. Destroying the object terminates monitoring. At construction time, you can specify the name of the output file, to ignore old blocks, to zap memory, and the maximum depth of stack crawls.

```
class TLocalHeapMonitor
{
  public:
    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);

    virtual           ~TLocalHeapMonitor();
};
```

TLocalHeapAnalyzer

`TLocalHeapAnalyzer` begins monitoring and analysis of a TStandardMemoryHeap when a TLocalHeapAnalyzer object is constructed. Destroying the object terminates monitoring. At construction time, you can specify the name of the output file, to ignore old blocks, to show only garbage, to zap memory, and the maximum depth of stack crawls.

```
class TLocalHeapAnalyzer
{
  public:
    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);

    virtual          ~TLocalHeapAnalyzer();
};
```

Heap monitor classes

These are the heap monitor classes.

TBlockEvent

`TBlockEvent: public MCollectible` represents one of three possible occurrences in the heap: a block allocation, a block deletion, or the registration of a pre-existing block. The last type of event is needed because when watching starts, there are blocks in the heap already for which no context information is known.

Block events have the following state information:

- The thread in which the event happened, a TSurrogateTask.
- The time of the event, a TTime.
- The first byte address of the block in question, a `void *`.
- The length of the block in bytes, a `size_t`.
- The object's v-table pointer, if any, a `void *`. This field does not exist for allocation events because newly-allocated blocks contain garbage.
- The call chain of the event, a TCallChain. This call chain is limited to a maximum frame depth which is a constructor parameter to TBlockEvent.

TBlockEventHandler

`TBlockEventHandler: public MCollectible` is an abstract base class that defines protocol for classes that process block event information. Such classes might, for example, write block event data to a text file, or use the block events to maintain a dynamic model of the heap's state.

```
class TBlockEventHandler : public MCollectible {
public:
    // Framework methods. These will be called in the order: HandleInitialize,
    // HandleBlockEvent (for each event), HandleFinalize.
    virtual void HandleBlockEvent(const TBlockEvent&, TAddressPeeker&) = 0;
    virtual void HandleInitialize(TAddressPeeker&) {};  // Override if desired
    virtual void HandleFinalize(TAddressPeeker&) {};    // Override if desired
};
```

TAddressPeeker

`TAddressPeeker` allows you to perform address-space-specific operations from another team. It maps addresses to symbolic names (function names), finds destructor addresses of objects, and reads the contents of memory in the remote address space. Use this class whenever *freezing* occurs in order to convert addresses in the target team into text symbols.

TAddressPeeker caches function names it finds, under the assumption that the same address will be looked up frequently. Because of this caching, instances of TAddressPeeker should be shared; that is, if multiple clients on a team need its services, they should share a single instance of it.

TAddressPeeker uses a TTeamHeapMonitor, a client of TTeamHeapMonitorDispatcher, to do its work.

```
class TAddressPeeker {
// This is not an MCollectible. Do not copy it, assign it, stream it, clone it,
// or do any other MCollectible operations.
public:
    TAddressPeeker(TTeamHeapMonitor* aliasedMonitor);
    virtual ~TAddressPeeker();

    // Functions in remote address space. These methods are multithread-safe.
    // DescribeFunction returns an unmangled function name, which it also caches
    // for subsequent calls. DescribeCallChain returns a tab-delimited list of
    // function names for a call chain. Describe object returns the class name
    // of an object on the heap.

    const TText& DescribeFunction(const void* address);
    const TText& DescribeCallChain(const TCallChain&);
    const TText& DescribeObject(const void* address);
    const TText& DescribeVTable(const void* address, const void* destructor=0);
    void CopyMemory(void* localCopy, const void* address, size_t bytesToCopy);
};
```

Heap analyzer classes

These are the heap analyzer classes.

THeapBlock

`THeapBlock: public TAbstractHeapBlock` represents a single block within a THeapMirror. It has state: live or frozen, allocated or deleted, first byte address, size, time of allocation, age, allocation context and deletion context, dynamic type. If it is frozen, the context information is flattened to text; otherwise, it consists of addresses in the target teams address space.

```
class THeapBlock: public TAbstractHeapBlock {
public:
    // Construct from the allocation event. Normally this is a allocation
    // or an already exists event. This can also be a deletion event, which
    // is anomalous, but will be handled correctly.
    THeapBlock(const TBlockEvent&);

    // Canonical methods
    THeapBlock(const THeapBlock&);
    THeapBlock& operator=(const THeapBlock&);
    virtual ~THeapBlock();
    virtual TStream& operator>>= (TStream& towhere);
    virtual TStream& operator<<= (TStream& fromwhere);
    MCollectibleDeclarationsMacro(THeapBlock);

    // Deletion
    void Delete(const TBlockEvent&);    // Delete using the block event
    void DeleteAnomalous();             // Mark as deleted; we never got deletion event!

    // Characteristics
    // Boolean IsDeleted() const;       // Inherited
    // void* GetAddress() const;        // Inherited
    size_t GetSizeInBytes() const;
    TTime GetAllocationTime() const;
    TTime GetAge() const;               // computes on the fly if needed

    void GetClassName(TText&);          // dynamic type: THIS IS USELESS AT THIS POINT

    // Describe
    void DescribeBlock(TAddressPeeker&, TText&) ;       // address size type age
    void DescribeAllocation(TAddressPeeker&, TText&) ;  // thread time stackcrawl
    void DescribeDeletion(TAddressPeeker&, TText&) ;    // thread time stackcrawl

    void Describe(TAddressPeeker&, TText&, UniChar separator);

    // Context
    const TCallChain* GetAllocationContext() const;
    const TCallChain* GetDeletionContext() const;
    TSurrogatethread GetAllocationThread() const;
    TSurrogatethread GetDeletionThread() const;

    // Freezing
    Boolean IsFrozen() const;
    void Freeze();
};
```

TAbstractHeapBlock

`TAbstractHeapBlock: public MOrderableCollectible` represents a block on the heap, and is an abstract base class descending from MOrderableCollectible. It has only two pieces of state data: its address and whether or not it is deleted. The latter defines canonical comparison methods (IsEqual, IsGreaterThan, IsLessThan) based on the address and deletion status. This allows you to search for a block in a collection at a certain address—either a deleted or a live block.

MHeapDiscipliner

`MHeapDiscipliner: public MCollectible` is an abstract class that defines the protocol for classes that monitor the correctness of heap behavior and usage. The method CheckBlockEvent verifies that the given block event is valid on the given heap model. If there is a pre-existing block at the address of the event, it is passed in. If CheckBlockEvent detects a problem, it creates a new THeapAnomaly on the heap and returns it; otherwise it returns 0.

```
class MHeapDiscipliner : public MCollectible {
public:
    virtual THeapAnomaly* CheckBlockEvent(const THeapMirror& heapBeforeTheBlockEvent,
const THeapBlock* preExistingBlockOrNil, const TBlockEvent& newEvent) = 0;
    VersionDeclarationsMacro(MHeapDiscipliner);
};
```

THeapMirror

`THeapMirror: public MCollectible` is a model that mirrors the contents of the heap. It maintains a sorted list of all blocks in the heap. When a block is deleted, it keeps the block in the model until a new block is allocated at the same address. This allows discipliners to differentiate a double deletion from a deletion of a non-block address. In fact, the mirror maintains deleted blocks until a new block is allocated and deleted. the mirror can store up two blocks at the same address: the last block that was deleted, and the current *live* one.

 NOTE Differentiating a double deletion from a deletion of a non-block address is insufficient in some cases. An example of this is if you allocate block #1, then delete it, then allocate block #2 at the same address, then delete it, then allocate block #3 at the same address, then delete a pointer to block #1. This final deletion will be incorrectly reported as a double deletion of block #2 rather than of block #1.

The heap model has data for each block in the heap. It maintains state information for the heap as a whole: whether it is live or frozen. If live, the heap has pointers into the team being watched and also contains an anomaly list.

If the heap is live, it is connected to a team, and contains a TAddressPeeker which allows it to resolve addresses to symbolic names and retrieve memory contents.

```
class THeapMirror: public MCollectible {
public:
    // Canonical methods
    THeapMirror();
    THeapMirror(const THeapMirror&);
    THeapMirror& operator=(const THeapMirror&);
    virtual ~THeapMirror();
    virtual TStream& operator>>= (TStream& towhere);
    virtual TStream& operator<<= (TStream& fromwhere);
    MCollectibleDeclarationsMacro(THeapMirror);

    // Accessing blocks
    TIterator* CreateBlockIterator() const;         // Exception on failure
    THeapBlock* FindBlockAt(void* address) const;
    THeapBlock* FindDeletedBlockAt(void* address) const;
    void AdoptBlock(THeapBlock*);
    void DeleteBlock(const THeapBlock&);
    long GetBlockCount() const;
    const TSortedSequence& GetBlocks() const;

    // Connect/disconnect
    enum EState {kConnected, kNotConnected, kBusy}; // Busy means in transition
    void ConnectToTeam(const TthreadHandle&, FrameCount maxDepth);
    void LaunchAndConnectToTeam(const char* teamName, FrameCount maxDepth);
    void Disconnect(Boolean freezeModel=TRUE);
    void WaitForDisconnect();

    // Information
    EState GetState() const;
    TthreadHandle GetTargetTeam() const;        // Invalid team if not connected
    void GetTeamDescription(TText&) const;

    void Describe();

    // Freezing, reset. Later may add Unfreeze(TthreadHandle).
    Boolean IsFrozen() const;
    void Freeze();
    void Reset();                               // Clear out all blocks and anomalies

    // Discipliners
    TIterator* CreateDisciplinerIterator() const;   // Exception on failure
    void AdoptDiscipliner(MHeapDiscipliner*);       // Adds discipliner to set

    // Anomalies
    TIterator* CreateAnomalyIterator() const;       // Exception on failure
    void AdoptAnomaly(THeapAnomaly*);               // Adds anomaly to list

    // Called by THeapMirrorAgent
    void IncorporateBlockEvent(const TBlockEvent&);
};
```

Connects the mirror to an existing, running team

Launches a new team and watches it until it dies

Dumps a text description of the heap to a text file

Updates the internal model, and also passes on the event to the attached set of THeapDiscipliner subclasses for checking

THeapAnomaly

`THeapAnomaly: public MCollectible` is an abstract class associated with THeapDiscipliner. Each THeapDiscipliner, if it finds an anomaly, constructs and returns a corresponding subclass of THeapAnomaly. The heap model maintains a list of such anomalies and displays them graphically. Anomalies can be connected to specific THeapBlocks.

Three concrete classes are TDeleteTwiceAnomaly, TDeleteNonBlockAnomaly, and TAllocTwiceAnomaly.

```
class THeapAnomaly: public MCollectible {
public:
    // Canonical methods
    THeapAnomaly(const THeapAnomaly&);
    THeapAnomaly& operator=(const THeapAnomaly&);
    virtual ~THeapAnomaly();
    virtual TStream& operator>>= (TStream& towhere) const;
    virtual TStream& operator<<= (TStream& fromwhere);

    // Description
    void SetDescription(const TText&);      // Subclasses should call in their ct
    void GetDescription(TText& toReceiveDescription) const;

    // Do not delete result of GetAssociatedBlock.
    THeapBlock* GetAssociatedBlock() const; // 0 if none

protected:
    THeapAnomaly(const THeapBlock* associatedBlock);
private:
    THeapAnomaly();                         // Disallowed; not defined
};
```

THeapMirrorException

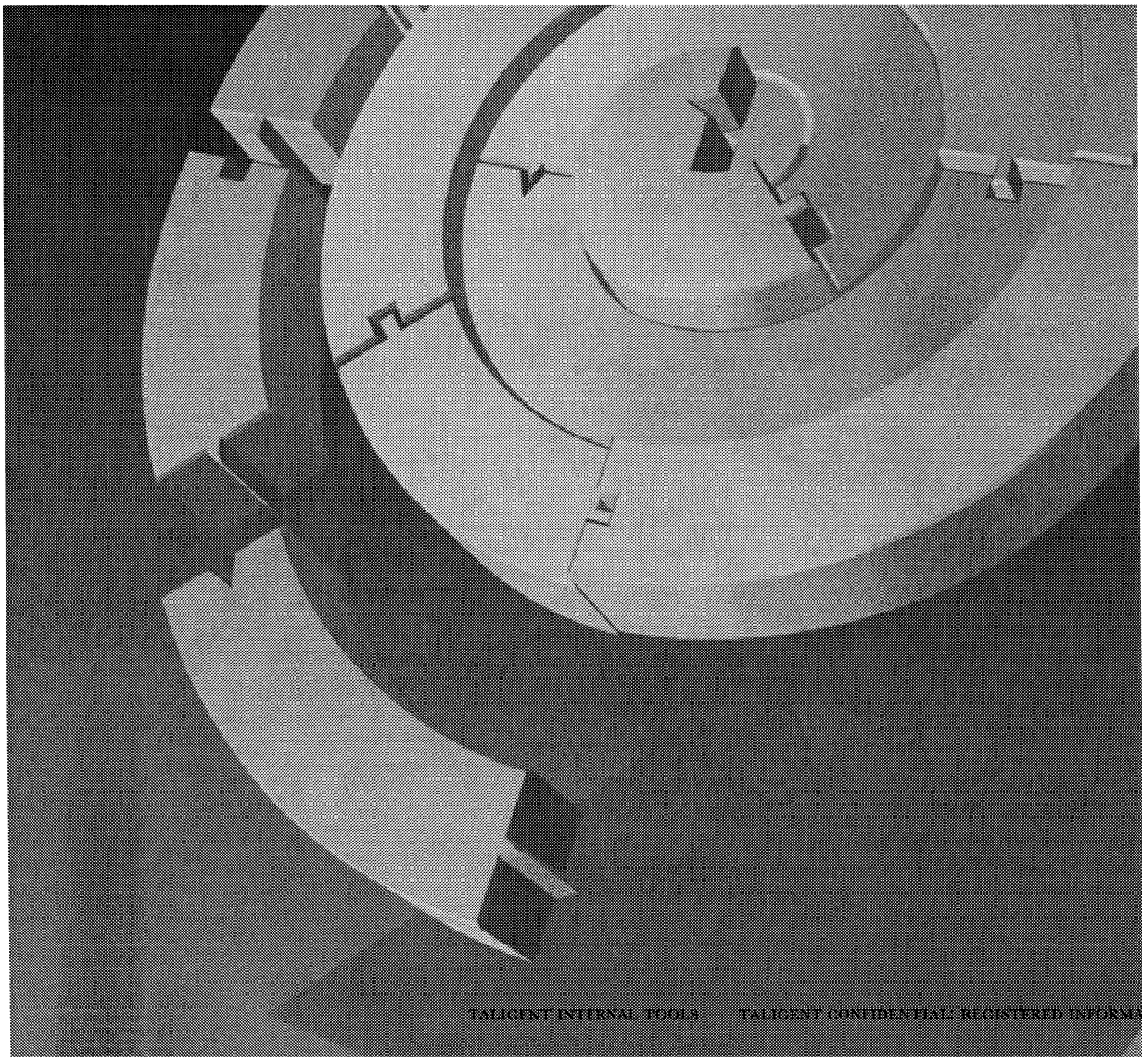
`THeapMirrorException: public TStandardException` is the exception class thrown by THeapMirror. It includes codes for invalid team, already watching team, not watching team, could not create iterator, block not found, and freeze without peeker.

Tool utility classes

Classes in this section are defined in the ToolUtilities project.

TCallChain

`TCallChain: public MCollectible` represents a call chain or stack crawl at a particular point of execution. It has methods that update its contents to reflect the current call point, and can skip some number of frames to get to the interesting part of the stack. TCallChain has a variable depth (it can grow or shrink at runtime) but it only changes its depth during copying or assignment. Otherwise it traverses the stack until its buffer is full. The size of this buffer is the maximum frame depth, and is a settable parameter in the heap tools.



CHAPTER 8

TEST TOOLS

The test tool included with Release A is TCL.

TCL

TCL 7.3 (pronounced "tickle") is a public domain scripting language from UC Berkeley. It is specifically written as a tool to integrate development tools. Its syntax is much like that of other UNIX shells (`csh`, `sh`, etc.), but it provides several specific advantages over the others, most notable is that it is highly extensible, portable, and embeddable inside of other programs.

Getting started with TCL is easy, but like any programming language, learning to take full advantage of its power takes time. Start with basic scripts, and learn more as you go.

NOTE This documentation is intended to get you up and running quickly as a tester using TCL. This is not a language reference or tutorial, but it does cover basic usage of the language for testing purposes.

The TCL shell—ttclsh

`ttclsh` is a variant of the standard `tclsh` provided with the TCL distribution (the extra "t" is for Taligent). This implements TCL, and should be available on your AIX system. `ttclsh` includes several important TCL extensions: `-[incr tcl]` (an object programming extension), Test Framework extensions, and (eventually) extensions to allow distributed script execution.

You can invoke `ttclsh` interactively, in which case it works much like `csh`. This can be handy when debugging scripts.

To run `ttclsh` on UNIX systems, all test script files should have execute permissions and have a first line that reads:

```
#!/usr/taligent/bin/ttclsh
```

On native systems, TCL will be the only scripting engine available, and this line will be ignored.

NOTE Test scripts should have a `.tcl` suffix so that its type of script is obvious.

Running tests from TCL scripts

Run tests from TCL by calling `tal_runtest`. `tal_runtest` is a wrapper for `runtest`; all `runtest` command options work with `tal_runtest`.

```
tal_runtest -echo d -t TMyTest MyTestLib
```

Before calling `tal_runtest`, start the Taligent AIX layer with `StartPink`.

Learning more about TCL

Like other UNIX scripting languages, TCL provides a great deal of support for complex scripts, including control flow structures, user-defined procedures, and local and global variables. Use these features when writing test scripts.

You can find out more about the features by reading the TCL documentation, which is available online or in printed form. The online documentation is in `man` pages. You can find useful TCL `man` pages in the `/usr/taligent/man/man1` and `/usr/taligent/man/mann`.

There are also raw PostScript™ documents in:

```
$TaligentRoot/Taligent/DevelopmentTools/Platform/AIX/tcl/docs
```

To get printed versions of these documents, request them from your Area Assistant. Though there is a 300 page book on TCL available for reference, due to copyright restrictions, we are unable to make copies for individuals. However, you can copy via FTP from:

```
harbor.ecn.purdue.edu: /pub/tcl/sprite-mirror/book*.ps.Z
```

These files can be uncompressed and printed on a LaserWriter™ printer using ShowPages, DropPS, or equivalent utility. Remember, though, that it is 300 pages!

NOTE If you do not have physical access to Taligent, you can obtain the TCL documents from the same ftp site.

Ensuring portability

TCL allows you to execute any UNIX command. However, do *not* use any commands other than built-in TCL commands in your scripts, because not all UNIX commands are portable to other systems. The *only* exception to this rule is launching Taligent Application Environment applications, because these should be available on any system you test.

Here are some commands to watch out for; do not use these commands in your scripts:

```
ls
cat
grep
awk
sed
perl
find
```

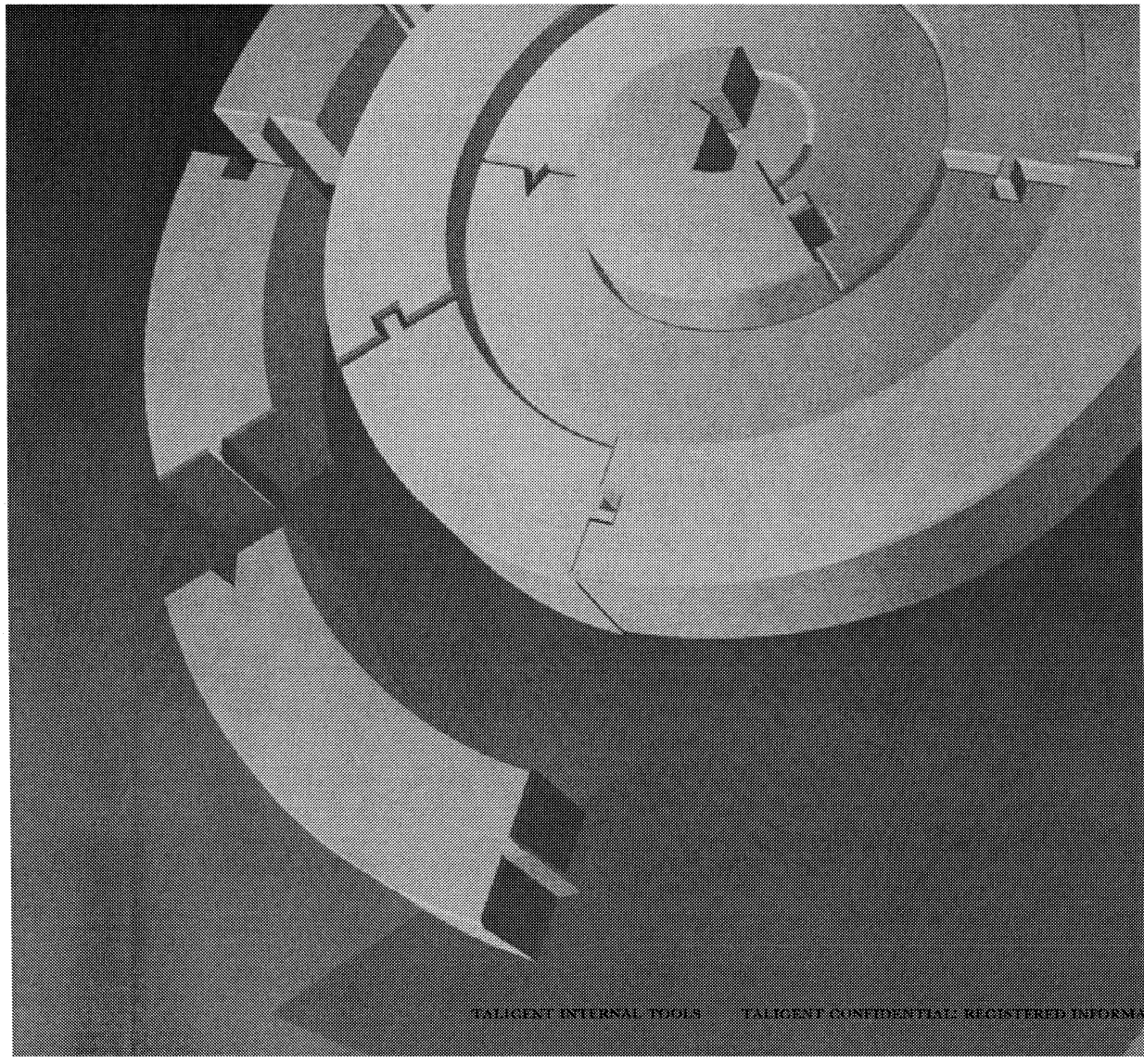
A TCL example

This example TCL script is an excerpt from the much larger Test Framework test script, but it is a complete example in and of itself. The only changes needed from the `sh` original were adding the header and doing a search and replace from `RunTest` to `tal_runtest` and from `echo` to `puts`:

```
#!/usr/taligent/bin/ttclsh
# $Revision:$
# +-------------------------------------------------------------------
# | Test script for TestTestFrameworkLib
# | Alan Liu
# | Copyright (c) 1992-1993 Taligent, Inc.
# |
# | This script relies on the libraries TestLib, BaseTestLib, &
# | TestTestFrameworkLib.
# | History:
# |    12/09/93        ET          Changed to shell script for AIX.
# |    12/20/93        AGS         Converted to tcl
# +-------------------------------------------------------------------

puts "### TestTestFrameworkLib.Script - Start..."

puts "### TNothingTest"
tal_runtest -log -t TNothingTest TestTestFrameworkLib
tal_runtest -log -e t -t TNothingTest TestTestFrameworkLib -n 100
```



CHAPTER 9

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 `xlC` 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:

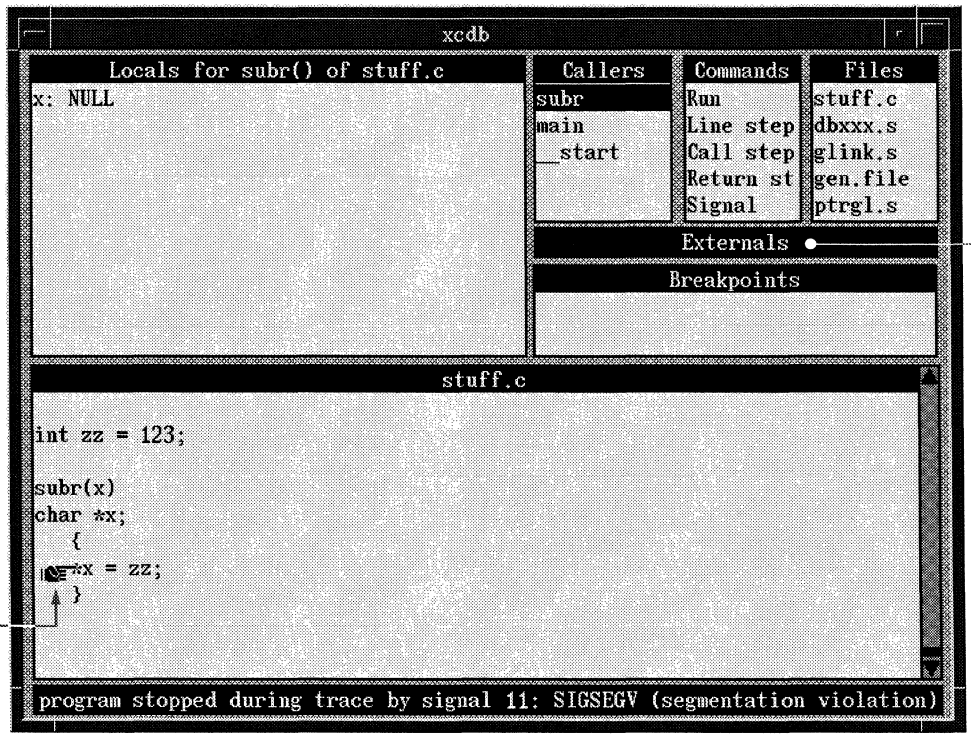
- Inspect the local environment of any function in the call chain and display the format (`signed`, `unsigned`, `hex`, etc.) of any individual variable
- Expand aggregate objects (`classes`, `structs`, `unions`, and `arrays`) to reveal arbitrary levels of detail
- Tailor window layout to your preferences 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

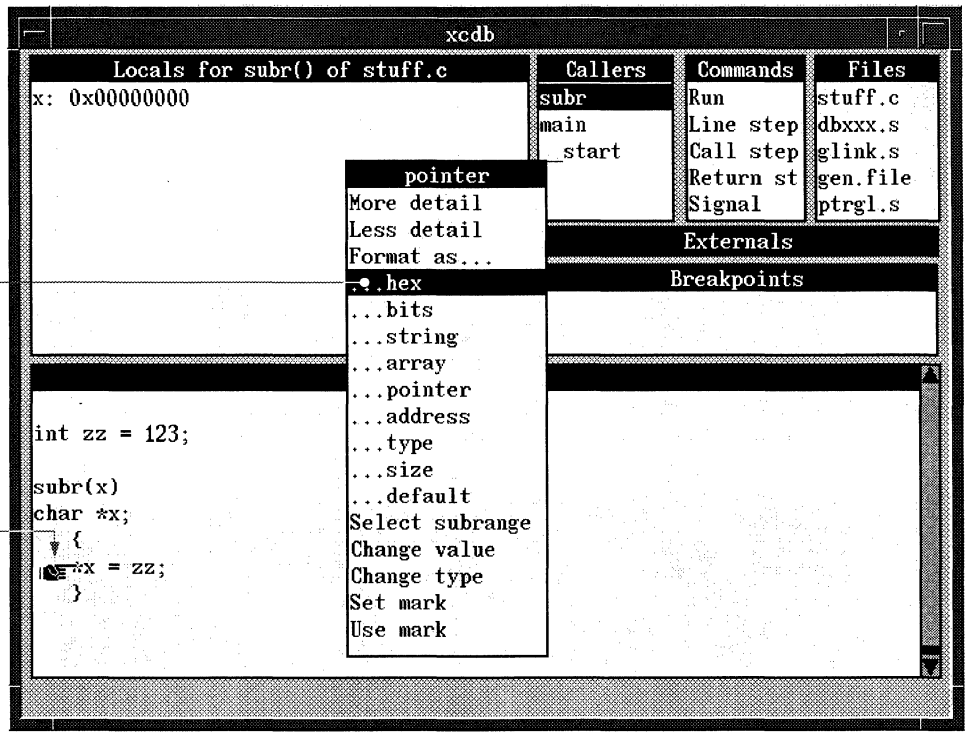
If the program interruption is of a type that allows execution to be continued, then you can resume program execution, perhaps after setting or clearing breakpoints. You can either ignore the signal that caused the interruption or pass it to the program.

Here is a typical display following a program exception.



The pointing hand icon marks the source line corresponding to the current instruction.

An iconized window. Left-click the icon to convert it to a normal window. Left-click the title bar to convert it to an icon.


xcdb
Locals for subr() of stuff.c
x: 0x00000000
Callers
subr
main
_start
Commands
Run
Line step
Call step
Return st
Signal
Files
stuff.c
dbxxx.s
glink.s
gen.file
ptrgl.s
Externals
Breakpoints
pointer
More detail
Less detail
Format as...
.hex
...bits
...string
...array
...pointer
...address
...type
...size
...default
Select subrange
Change value
Change type
Set mark
Use mark
int zz = 123;
subr(x)
char *x;
{
*x = zz;
}
Activate this menu,
by left-clicking the "x".

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.tarbin` as a binary file, and process it with the `tar`. For example, if you have `xcdb6000.tarbin` and 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 `.Xdefaults` file, by a command line parameter, or by the window manager. "Customization" on page 140 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

–geometry *WxH+X+Y*	A window size and position, overriding the specification in `.Xdefaults` (if any).
–font *fontname*	The name of a font, overriding the specification in `.Xdefaults` (if any).
–title *title*	A title to place on the window border.
–bw	Use a black on white color scheme.
–wb	Use a white on black color scheme.
`-I` *dirname*	A directory to search for source files which cannot be found in the current directory (multiple `-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 138.
–a *pid*	The ID of an existing process to attach to, instead of starting a new process.
–r *funcname*	Specifies how far to run the program's initialization routines. Normally the program runs to the symbol `main`, 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 `-r \verb,__,start`. To stop at the function which initializes C++ static objects, specify `-r \verb,__,C\verb,_,runtime\verb,_,startup`.

–e *numelts*	The maximum count of elements to display for any array (default is 1000).
–c *numcalls*	The maximum count of functions to display in the function call traceback (default is 20).
–d *numdetails*	The count of detail levels to add (or remove) when More(or Less) detail is selected from a data object formatting menu.
–b *numbreaks*	The maximum count of breakpoints that can be set simultaneously (default is 50).
–i *signo*	The number of a signal to ignore and pass to the program (multiple `-i` flags are cumulative).
–f *fetcher*	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.
–l	Write window layout information to a file named `sample-layout` when the debugger exits. You can then copy this file into your `.Xdefaults` file where it will be read when you next run the debugger. See "Customization" on page 140.
–q	Run *quietly*, only revealing the debugger if the program being debugged stops due to a signal or runtime exception.
–v	Run *verbosely*, print status information and commentary while running.
–n	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.
–p	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`) even if they were moved after the executable was generated.
program	The name of the program to execute.
args	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	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. 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 *out of lined* inline), a menu prompts you to choose the function instance to breakpoint.
Locals	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 130).
NonLocals	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 130).
Callers	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.
Functions	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.
Files	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.
Breakpoints	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 *stop sign* icons.
Command	Displays the commands which can be used to control the debugger. Click on command to execute it.
Messages	This window pane displays messages from time to time. It is invisible unless there is a message to see.

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	Enlargse 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 *pushes the window beneath* 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.
:*nnn*	Moves cursor to line number *nnn* (but not past end of file).
/*XXXX*	Search forward to next occurrence of the string *XXXX*; omit the *XXXX* to repeat search from current position.
XXXX	Moves cursor backward to preceding occurrence of *XXXX*; omit the *XXXX* 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	Invokes an editor on the file in the Listing window. Specifies the editor with `xcdb.Edit` in your `.Xdefaults`. Use `%s` and `%d` symbols for filename and line number, respectively. For example, to invoke `vi`:
	`xcdb.Edit: (xterm =+0-0 -n Vi -e vi +%d %s &)`
	To invoke `emacs`:
	`xcdb.Edit: (emacs '+%d' '%s' &)`
	To invoke `v`:
	`xcdb.Edit: (v -l %d %s &)`

Restart	Terminates the program, reloads it, and sets its execution point back to the beginning; all breakpoints and data format selections remain unchanged. If `stdin` is a file, it is rewound to start-of-file.
Exit	If the debugger was attached to a process using `-a`, then the process is allowed to resume execution (if you want the process to die, you must use `kill -9` from an xterm window—there's no explicit command to do this from `Xcdb`); otherwise, the process terminates and the debugger returns to the operating system.
Preferences	A menu prompts adjustments for `Xcdb`'s behavior. See"Preferences" on page 138.

[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 132).
	`array`	The word "array" (or an element list, see "Array formatting" on page 134).
	`pointer`	The word "ptr" (or a pointed-to object, see "Pointer formatting" on page 137).
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 `xcdb()` member function (if any). See "Self-displaying C++ objects" on page 139.
Array	More detail	Reveals array elements.
	Less detail	Hides array elements.
	String	Displays an *array of characters* 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 134.

Pointer

Less detail	Hides the pointed-to object.
Hex	The pointer in hex format.
String	A *pointer to character* as a null terminated string.
Array	At *pointer to X* as an *array of X*.
Select subrange	A selected subrange of the pointed-to array. A prompt asks for the elements you wish to see.
Cast	Changes (*casts*) 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++ *pointer to abstract base class* into a *pointer to most derived class* 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:

```
Node: node
Node: { NULL data }
Node:   next: NULL
        data:  data
Node:   next: NULL
        data:  { 1 123.000000 }
Node:   next: NULL
        data: type: 1
            value: 123.000000
Node:   next: NULL
        data: type: 1
            value: 123.000000
Node:   next: NULL
        data: { 1 123.000000 }
Node:   next: NULL
        data: data
Node: { NULL data }
Node: node
```

Click More detail *here*

Click More detail *here*

Click More detail *here*

Click More detail *here*

Click More detail *here*

Click Less detail *here*

Click Less detail *here*

Click Less detail *here*

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

```
Node: { NULL data  }
Node: { NULL { 1 123.000000 } }
Node: { NULL { 1 float } }
Node: { NULL { 1 0x42f60000 } }
```

Click More detail *here*

Click Type *here*

Click Hex *here*

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:

```
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
```

Click `String` *here*

```
stuff: ->"abc"
```

Click `Select subrange` *here, enter "0,2,..."*

```
stuff: { "abc" "def" NULL }
```

Click `More detail` *here*

```
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:

```
Table: array
```
Click More detail *here*
```
Table: { ptr ptr NULL }
```
Click More detail *here*
```
Table: 0: ptr
       1: ptr
       2: NULL
```
Click More detail *here*

Click Type *here*
```
Table: 0: -> main()
       1: ptr
       2: NULL
```
Click Type *here*
```
Table: 0: -> function-returning-int
       1: ptr
       2: NULL

Table: 0: pointer-to-function-returning-int
       1: ptr
       2: NULL

Table: 3-item-array-of-pointer-to-function-returning-int
```
Click Type *here*

BREAKPOINTS

Set or remove unconditional breakpoints by clicking on the line in the Listing window. Set or remove conditional breakpoints that releate 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.
   - A. 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 *row major* form.
	C++	Array element addresses are computed in *row major* form; variable names are *demangled;* nested class members are labeled.
	FORTRAN	Array element addresses are computed in *column major* 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 142.
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 `#include` file information appearing in the symbol table.
	Ignore	The debugger ignores `#include` file information appearing in the symbol table. This option is a work-around for bugs in `cpp`, `cc`, and `cfront`—see "Frequently asked questions" on page 142.
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 `-s`.	
Upon fork follow	Controls tracing of `fork()` system calls:	
	Parent	Follows the parent process after a `fork()`
	Child	Follows the child process after a `fork()`
	When stepping through a `fork()` 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/.Xdefaults`. 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 `.Xdefaults` 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: *geometry*	Main window size and placement
Font: *font*	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 123 for information.

SpecialLayout: yes | no	Do window specifications follow?
*xxxx*Geometry: *geometry*	Size and placement for normal window
*xxxx*IconGeometry: *geometry*	Size and placement for iconized window
*xxxx*IconifyOk: yes | no	Permit iconization of this window?
*xxxx*IconStartup: yes | no	Iconize window at start-up?
*xxxx*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: *color*	Window borders, mouse outside
	BorderActive: *color*	Window borders, mouse inside
Xcdb ignores color entries on monochrome displays or when `-bw` *or* `-wb` *has been specified.*	Foreground: *color*	Normal text
	Background: *color*	Normal text
	MouseBody: *color*	Mouse body
	MouseOutline: *color*	Mouse outline
	CursorForeground: *color*	Cursor
	CursorBackground: *color*	Cursor
	MarkForeground: *color*	Marked text
	MarkBackground: *color*	Marked text
	TitleForeground: *color*	Window pane titles
	TitleBackground: *color*	Window pane titles
	DialogForeground: *color*	Command lines
	DialogBackground: *color*	Command lines
	DimForeground: *color*	Non-selectable menu items
	DimBackground: *color*	Non-selectable menu items
	ScrollbuttonIdle *color*	Scroll buttons, mouse outside
	ScrollbuttonActive *color*	Scroll buttons, mouse inside
Other	Editor: *command*	The specified *command* is invoked when the Edit command is selected from the Commands window (see earlier).
	Language: *language*	The debugger's behavior is adjusted for the specified *language*, as described in the Preferences menu section (see earlier). *language* must be one of: ▪ C ▪ C++ ▪ FORTRAN
	RespectIncludeFiles: yes | no	Controls interpretation of file symbols appearing in the symbol table, as described in the {\it Preferences} menu section (see earlier).
	ArrayLimits: *NNNN*	Controls data formatting, as described for the "–e" command line flag (see earlier).
	DetailPerClick: *NNNN*	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 xlC).

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 `xlC` (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 `xlC` 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 `xlC`, 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 140).

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:

1. Format the first item in the array
2. 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 `.Xdefaults` 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`?

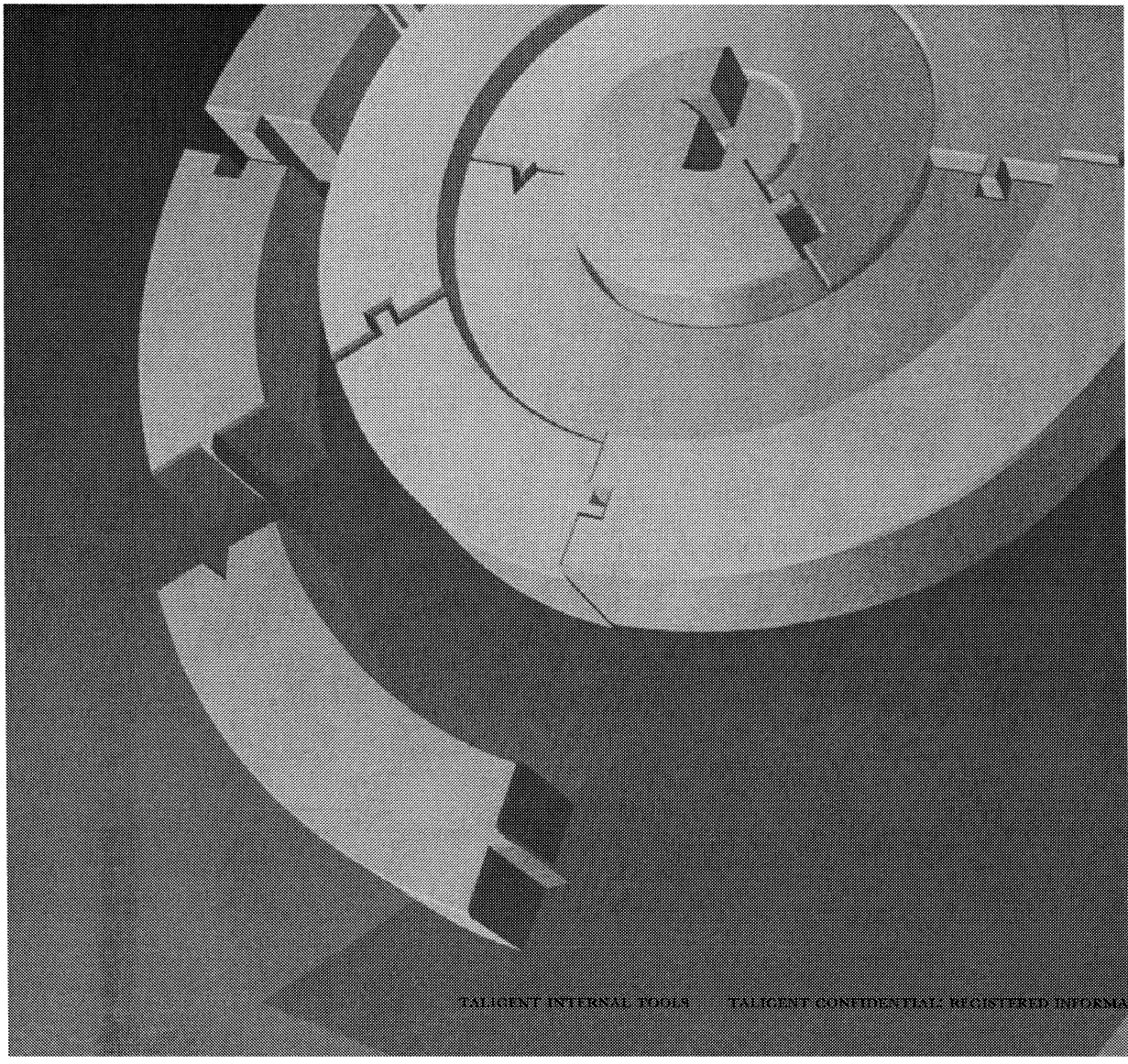
A: Please read the XCDB6000 NEWS file that is shipped with each XCDB6000 PACKAGE.

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.



CHAPTER 10

GDB

To debug the Taligent Operating System, use the GNU Debugger (GDB). This chapter help you use the GDB debugger quickly. For more detailed information, refer to the GDB Reference Manual. For information about debugging Taligent Application Environment, see Chapter 9, "Xcdb."

INSTALLING GDB

Internal Note	The installation procedures have not been finalized at this time.

RUNNING GDB

To debug *MyProgram* (a `runpink` executable):

1. Place *MyProgram.Herbie* in the source-code directory, if you want symbols.

   You can also specify the symbol-table file with the `-se` command-line option.

2. At the UNIX shell prompt, run `gdb` and specify the name of the program to debug:

```
% gdb MyProgram

GDB Taligent Version 4.11.D7 (rs6000-ibm-aix3.2),
Copyright 1993 Free Software Foundation, Inc.
(gdb)
```

The GDB command prompt

GDB is now ready to start debugging.

NOTE For information about `rp` executables, see "Debugging shared libraries and rp-executables" on page 154.

Source-level debugging

By default, GDB looks for source-level debugging source-code in the current directory. To specify another directory, use `directory` or the `-cd` command-line option:

```
(gdb) directory /home/Work/Taligent/Development/Portable/Albert/Source
```

To search multiple directories, use `dir`, or the `-d` command-line option, to add a directory to the search path:

```
(gdb) dir /home/Work/Taligent/Development/Portable/Albert/Test
```

To report which directories are in the search path, use `show dir`:

```
(gdb) show dir
```

Executing programs

To start you program, use `run`. If the program needs command-line arguments, include them on the `run` command line:

```
(gdb) run arg1 arg2
```

You can also use `set args` to set command-line arguments, and `info args` to find out what they are.

Getting help

To get online help, use `help`. For help on a specific topic, specify the topic.

```
(gdb) help breakpoints
```

Quitting GDB

To quit GDB, use `quit`. If you need to quit while a program is running (and you can't get to the GDB prompt), use Ctrl-C.

USING BREAKPOINTS

To set a breakpoint, use `breakpoint` (which you can abbreviate to `b`). To break when entering a function, specify the function name:

```
(gdb) b myfunc
```

To break at a particular line number in a specific file:

```
(gdb) b mysourcefile.C:228
```

To break at a specific address:

```
(gdb) b *0xdef00000
```

C++ mangles the names of member functions. To choose the member function from a list, specify the class and function name, followed by a tab character:

```
(gdb) b 'fooclass::foofunc<TAB>
```

NOTE Breakpoints can have a pass count or a condition, and you can execute commands after a breakpoint occurs. Type `help break` for more information.

Resuming execution

To resume program execution, use `continue`. To step command-by-command, or until a specific event, follow the instructions in the next section.

USING STEPS

To step between function calls or machine instructions, use these commands:

`step`	Steps one source code line.
`next`	Steps over a function call.
`stepi`	Steps one machine instruction.
`nexti`	Steps one machine instruction, stepping over function calls.

All of the variants of `step` and `next` accept a pass count, for example to step twenty lines at once:

```
(gdb) step 20
```

For more specialized control:

`finish`	Finishes a function.
`until` *`linenumber`*	Runs until linenumber. Be careful, if your program doesn't reach this line number, your program can hang.
`return` *`value`*	Forces a return with the optional given value.
`goto` *`label`*	Forces a goto in the program you are debugging.

EXAMINING DATA

GDB has several commands for examining data.

`print` *`expression`*	Prints, in a formatted manner, the value of an expression.
`x` *`address`*	Examines (or dumps) a memory address.
`memberfunc`	Toggles printing of member functions.

TRACING INSTRUCTIONS

Taligent's version of GDB has an instruction-trace facility. To use it, set up GDB in the usual way, and run to the breakpoint where you wish to start tracing. Then, put your trace into a file with `outfile` and begin the trace:

```
(gdb) outfile MyTracefile
```

You can trace to an end of function, a discrete number of instructions, or a particular address:

```
(gdb) trace
(gdb) trace 500
(gdb) trace 0xd1c40000
```

When you are finished, close the output file by calling `outfile` with no arguments:

```
(gdb) outfile
```

DEBUGGING SHARED LIBRARIES AND RP-EXECUTABLES

To debug shared libraries and `rp`-style executables:

1. Make sure you have these files:
   - The `rp` program and one extra terminal session.
   - The shared library version of `LLSystemLib` (the nonshared library version is `LLSystem.lib`).
   - `LLSystemLib.Herbie` and your target program's `.Herbie` file in the same directory where you run `LLSystemLib`.
2. Start the library server in the extra terminal session, this becomes the *libserver* session:

   ```
   rp _libserver
   ```

3. Run GDB and `LLSystemLib` in your original session, this is the *gdb* session:

   ```
   gdb LLSystemLib
   ```

   Once the `(gdb)` prompt appears, `LLSystemLib` is in memory, _StaticDataInit has been called, and you can set breakpoints in LLSystemLib.

4 Use `run` to execute the program:

```
run MyProgram arg1 arg2
```

Before running your program, `LLSystemLib` calls GDB's LibraryLoadedCallBack function, which causes GDB to load `MyProgram.Herbie` to retrieve the symbolic information.

NOTE As it is currently enabled, GDB's LibraryLoadedCallBack function causes a break into the debugger so that you can set breakpoints in your program before it runs. (Eventually there will be a mechanism similar to the 68K ci's `run -d` option.) When this break occurs, GDB prompts something like this:

```
Reading symbols:  your_program.Herbie
Symbol base at 0x20400108
LibraryLoadedCallBack:  Doing breakpoint
Gdb selected thread NN

Program received signal SIGTRAP (5), Trace/BPT trap
0xab8c4 in ?? ()
```

LibraryLoadedCallBack is defined not to have symbolic information display for itself, hence the hex address. Use `where` to see the `LLSystemLib` stack trace.

Reporting shared library symbol tables

To list all shared library symbol tables loaded, use `info shared`:

```
(gdb) info shared
```

PROBLEMS AND OTHER USEFUL INFORMATION

If you encounter a bug in GDB, file a Proteam bug report—GDB should soon be a component on the bug tracking system. meanwhile, here are some known problems and remedies:

- Your copies of `gdb` and `gdbnub` must be checked with the `blessit` script. As the root user, run `blessit` on both files. Once these files have been checked, `ls -l` shows the following permissions and ownership:

```
-rwsrwxr-x    1 root     system  ...
```

- Your machine should be running `snames` and `machid`. Run `ps -A | grep snames` to see if it has been set up to do so.
- Variable and type information is not currently supported. Use the `x` to examine memory.
- When you are using `LLSystemLib`, attaching to a program that is blocked in a system call works as far as getting the current user state, but the program aborts if you `step`, `continue`, or `detach`.

Internal Note Got any clues as to why?

- If you don't get a complete stack crawl when running `LLSystemLib` or multiple threads, run `tl`, and try it again. It usually works the second time.
- Don't restart a program in the same GDB session once the program terminates or is already running. Instead, quit and restart GDB and your program.
- Sometimes when continuing from a breakpoint, the program seems to hang. If you Ctrl-C to interrupt it, GDB shows that it didn't continue from the breakpoint.

- The Debugger call currently isn't provided in `LLSystemLib`. For your program to give control to the debugger, execute an `int 3` breakpoint instruction. One way to do this is to link an assembly module, `Debugger.s`, to your program:

```
Debugger    function    extern
            char        0xcc
            ret
            end
```

  Assemble it as follows:

```
Assembler.x86 Debugger.s
```

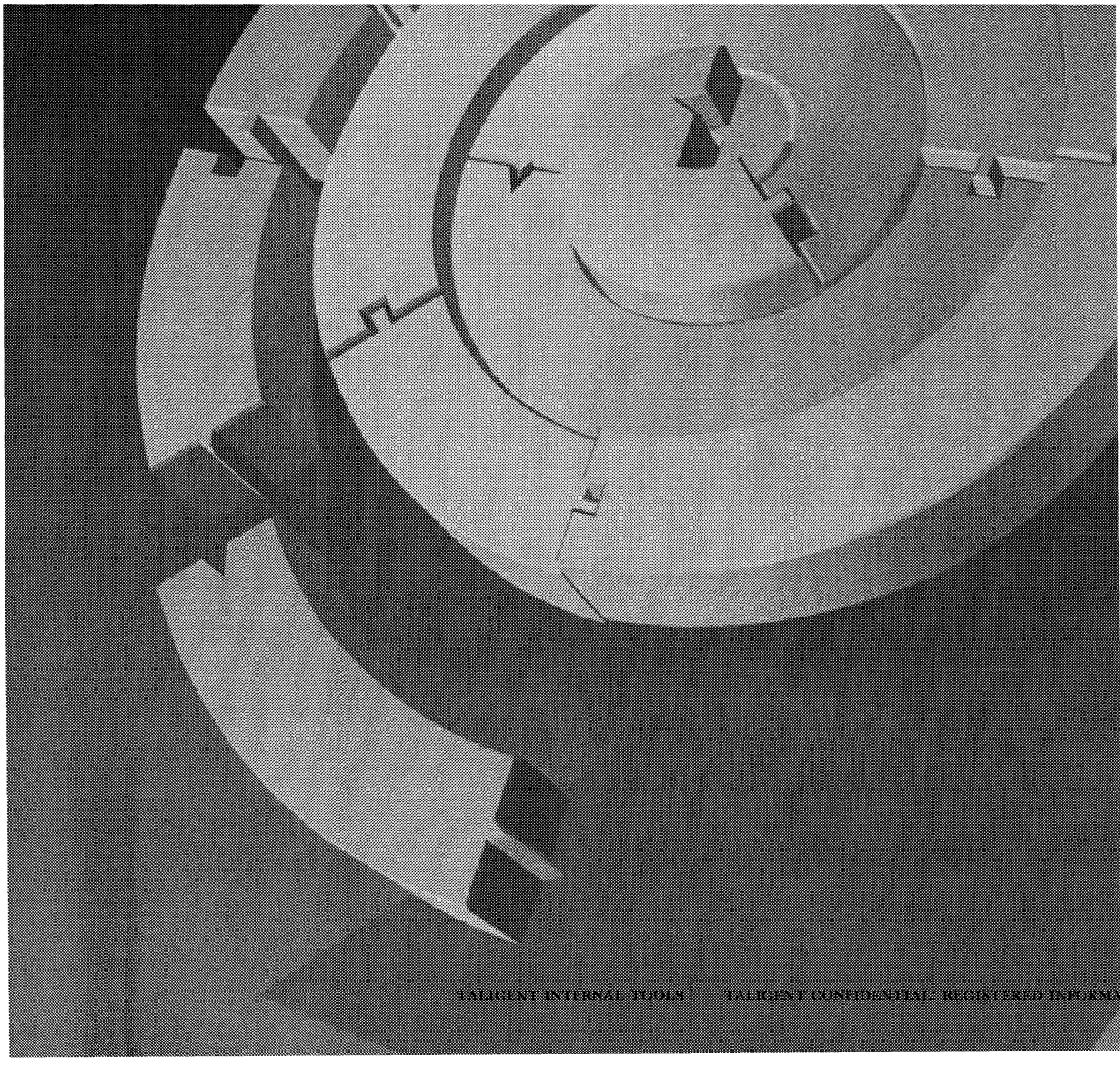
- The current version has some thread support. The thread commands for Mach conflicted with the standard GDB thread commands, so Taligent changed them by modifying the command prefixes to `mthread` and `mtask` (`help mthread` and `help mtask` reveal which commands are available). Where a thread argument use the MID, not the slot number.

  The following aliases are defined:

`mth`	`mthread`	The prefix
`tl`	`mthread` *`list`*	Display thread list
`ts`	`mthread` *`select`*	Select a thread

  GDB has the concept of a *current thread* that determines which registers and stack is displayed. The current thread is initially the thread where the breakpoint is hit. `ts` and `mthread select` specify a different thread to be the current thread. `tl` and `mthread list` report the known threads. Use the MID number to specify threads in all the commands. In the thread list, the current thread is marked with an "*" following the MID.

- If you step inside a function with no source map information GDB might give you a message like "Cannot access memory at address 0," or some message about not knowing the size of the function. Some of this will eventually be fixed. For now, use one of the following:
  - `si` (perhaps with `display`) to assembly step
  - `finish` to go to the end of call
  - `frame` to change to a frame with source and then set a breakpoint following the call you are in



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.

This chapter assumes you have the standard AIX environment. Make sure you set up your account to the specifications defined in Chapter 2, "Working in the AIX environment."

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 9, "Xcdb," before using this debugger. However, there are two things that can make your work with `xcdb` easier:

- Use `SCMFetch` to checkout sources from the SCM source data base (for more information, see"SCMFetch" on page 34).
- 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.

Within the Taligent Operating System, `OpusBug()` prints a debugging string, and then calls the debugger.

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

When you call `OpusBug()` within the Taligent Operating System it prints a message, and drops directly into the debugger.

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.
#

PROCESS_ID=$1
PROGRAM_NAME=$2

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
```

Arguments from OpusBug()

Calll the debugger

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
```

EMACS

Many engineers use the Emacs editor on AIX. This section details certain major aspects of Emacs that you might find useful if you use the editor.

Emacs shell

AIX has built-in terminals (*AIXTerms*) that most users do not need or like. As such, users tend to prefer using Emacs in shell mode, which is similar to an MPW Worksheet because you can scroll back, edit, and re-execute commands (unlike in AIXTerm).

Create a new shell

To create an emacs shell session:

1. Press ESC-x
2. Type `shell`

Create multiple shells

To have multiple shells sessions open, first rename your open shell buffer:

1. Press ESC-x
2. Type `rename-buffer`

Then create another new shell.

Emacs function keys

This section provides detail about each function key including, how and when you can use them. Each table includes the key, the command that Emacs executes, and a brief description of the action.

On page 168, you will find a function key quick reference to post next to your workstation.

F1—Build

Use F1 to build your subsystems. Emacs calls `Makeit` to execute a build (see page 66 for `Makeit` specific information). You can type these commands in the shell buffer, but it is better to use F1 because Emacs redirects the output of the build is **compilation** buffer, which is used by the error finding key, F2.

F1	`Makeit [target]` By default, F1 builds Includes, Objects, Exports, and Binaries for the current project and all its subprojects. You can change the default *target* by typing `new target`.
Ctrl-F1	`Makeit Clean Complete` Executes a clean build on the current project and all its subprojects. If you change these targets, they will not be remembered next time you use C-F1.
S-F1	`Makeit -c [target]` Builds the specified target in the current project only. Emacs remembers *target* after you first type it in. This is useful when you are building the same application over and over.

Meta-F1	`Makeit Binaries` Builds the binaries target for the current and all it's subprojects. This is useful for rebuilding shared Libraries. If you change these targets, Emacs does not remember them the next time you use Meta-F1.
Meta-Sh-F1	`Makeit -c Binaries` Builds the binaries target for the current project. This is useful if you have already built a single subproject, and you want to rebuild a shared library that is built in the parent directory.
Ctrl-Sh-F1	`Makeit -c Clean Complete` Executes a clean build on the current project only. If you change these targets, Emacs does not remember them the next time you use Ctrl-F1.

F2—Locate compiler messages

Locates errors or warnings generated during a compile, and which are in the **compilation** buffer. To get the compiler messages in this buffer, use F1.

F2	`Locate Next Message` Opens the file to the location that the compiler message refers to. This key finds (W)arnings, (E)rrors that the compiler fixed, and (S)erious errors that break the build. Note: you can use F2 to locate the result of a search, see below.
Sh-F2	`Locate Next Serious Error` (*Not implemented yet*) Finds the next (S)erious compiler error, and skip the(E)rror and (W)arning messages.
Meta-F2	`Locate Next Error Message` (*Not implemented yet*) Finds the next (S)erious or (E)rror compiler message, and skip the (W)arning messages.

Internal Note Are the keys implemented yet?

NOTE Theses keys might not work with the Comptech compiler.

F3—Search

Searches (grep) for patterns in specific locations, and redirects the result to a special buffer that F2 can use locate the match.

F3	Searches TaligentIncludes Prompts you for a pattern and then searches for that pattern in `$TaligentIncludes`, `$TaligentPrivateIncludes`, and then your local project, in that order.
Sh-F3	Searches Current project (*Not implemented yet*) Prompts you for a pattern and then searches for the pattern in your local project.
Meta-F3	General purpose search Prompts you with a grep (search) command for general purpose searches. Unlike a terminal, the result goes to a special buffer so you can view the matches with F2.

F4 —Taligent AIX layer

Starts and stops the Taligent AIX reference layer, and a Taligent application within that layer.

F4	Starts the AIX Layer Eexecutes `StartPink` to start the AIX reference layer.
Sh-F4	Stops the AIX Layer Executes `StopPink` to stop the AIX reference layer. Does *not* kill applications that are running; you must close those application first.
Ctrl-F4	Starts a Taligent application Launches the specified application. Emacs remembers the application name after the first time you type it in.
Meta-Sh-F4	Starts a Taligent application with `xcdb` Launches the specified application using the `xcdb` debugger. Emacs remembers the application name after the first time you type it in.

NOTE Theses keys might not work with the Comptech compiler.

F5—Goto

Goes to and reports your location in the buffer.

F5	GoTo line Prompts you for a line number, and then takes you to that line in the current buffer.
Ctrl-F5	What line? Prints the cursor's line number to the status line.
Sh-F5	GoTo Help Brings up a buffer with a quick reference to all the function keys.

F6 and F7—Change buffers

Switches your current buffer. There are many other ways to change your current buffer, but this makes it easy.

F6	Previous buffer Brings the last buffer you visited (before current one) to the current buffer.
Sh-F6	Burry buffer Puts the current buffer last in the list of buffers and brings the front most buffer in the list to the front. Think of this as a buffer que.
F7	Next buffer Bring.
Sh-F7	Unburry buffer Bring.

F8—Open

Opens the current selection, or open the .PinkMake for the current project.

F8	Opens current selection Open a file by the name of the current selection. Searches `$TaligentIncludes`, `$TaligentPrivateIncludes`, and the current directory. Useful in opening header files by select the filename on the `#include` line.
Sh-F8	Opens Current PinkMake Opens the .PinkMake file for the current project.
Ctrl-F8	Comments-out current selection Put C++ style comments at the beginning of each line in the current selection.

F9, F10, and F11—Keyboard macros

Defines and runs keyboard macros. Macros are useful when you have to perform repetitious editing tasks. If you find yourself running a sequence of commands over and over again, it might be efficient to define a macro for the commands.

F9	Start keyboard marco Begins recording commands and key strokes.
F10	End keyboard macro Stop recording.
F11	Call last keyboard macro Execute the last keyboard macro recorded.

To name the last macro to save it for later use (and not record over it):

1. Press ESC-x
2. Type `name-last-kbd-macro` and press Return.

   Emacs then prompts you for the name.

F12—Find and replace

Searches and replaces strings.

F12	Query replace string Emacs prompts you for the search string, then prompts you for the replacement string. Emacs then moves your insertion point to the first occurrence of the string and prompts you about replacing the string. It then goes to the next occurrence.
Sh-F12	Replace string Does a global search and replace; Emacs prompts you for the search string, and then prompts you for the replacement string.

F13 (Print Screen)—Checkout

Checks out the current buffer for modification. A second buffer shows the `Checkout` information.

F14 (Scroll Lock)—Checkin key

Checks in the current buffer to its RCS project. A second buffer shows the `Checkin` information.

Navigation keys

Here is a quick reference of navigation keys:

Home	Move cursor to beginning of line
End	Move cursor to end of line
Ctrl-Home	Move cursor to beginning of buffer
Ctrl-End	Move cursor to end of buffer
Strl-Home	Move cursor to beginning of window
Sh-End	Move cursor to last line of window
PageUp	Scroll down
PageDown	Scroll up
Ctrl-PageUp	Scroll other window down
Ctrl-PageDown	Scroll other window up

Emacs and tags

Tags are helpful in finding class definitions and member functions. Taligent AIX layer system builds have a TAGS file for the Taligent include files. The standard Emacs configuration file (`current cpg.el`) automatically loads the TAGS file.

To use tags, place the cursor over the class or member function to look up, and press Esc-. (Esc-period). Emacs opens the file where that class or member function is defined. Esc-, (Esc-comma) finds the next occurrence of the tag.

 NOTE The Taligent Operating System environment does not build these tag files.

EMERGE

Emerge is a set of Emacs macros that merge two `diff` files (the result of comparing three source files).

Commands

In *edit* mode, you must use C-c or C-c to begin commands; you can use commands in *fast* mode without the prefix.

Key	Binding	Key	Binding
C-]	emerge-abort	-	negative argument
.	emerge-find-difference	0--9	digit argument
<	emerge-scroll-left	>	emerge-scroll-right
^	emerge-scroll-down	a	emerge-select-A
b	emerge-select-B	c	Prefix Command
d	Prefix command	e	emerge-edit-mode
f	emerge-fast-mode	i	Prefix Command
j	emerge-jump-to-difference	l	emerge-recenter
m	emerge-mark-difference	n	emerge-next-difference
p	emerge-previous-difference	q	emerge-quit
s	Prefix command	v	emerge-scroll-up
x	Prefix command	|	emerge-scroll-reset
c b	emerge-copy-as-kill-B	c a	emerge-copy-as-kill-B
d b	emerge-default-B	d a	emerge-default-A
i b	emerge-insert-B	i a	emerge-insert-A
s s	emerge-skip-prefers	s a	emerge-auto-advance
x x	emerge-set-combine-versions-template	x t	emerge-split-difference
x s	emerge-split-difference	x m	emerge-set-merge-mode
x l	emerge-line-numbers	x j	emerge-join-differences
x f	emerge-file-names	x C	emerge-combine-versions-register
x c	emerge-combine-versions	x l	emerge-one-line-window

Modes

Emerge has several modes of operation.

Fundamental mode is used for comparison with the other modes.

Emerge mode minor mode (indicator is *emerge*) is used by Emerge when merging files, and can be entered through one of the functions:

```
emerge-files
emerge-files-with-ancestor
emerge-buffers
emerge-buffers-with-ancestor
emerge-files-command
emerge-files-with-ancestor-command
emerge-files-remote
emerge-files-with-ancestor-remote
```

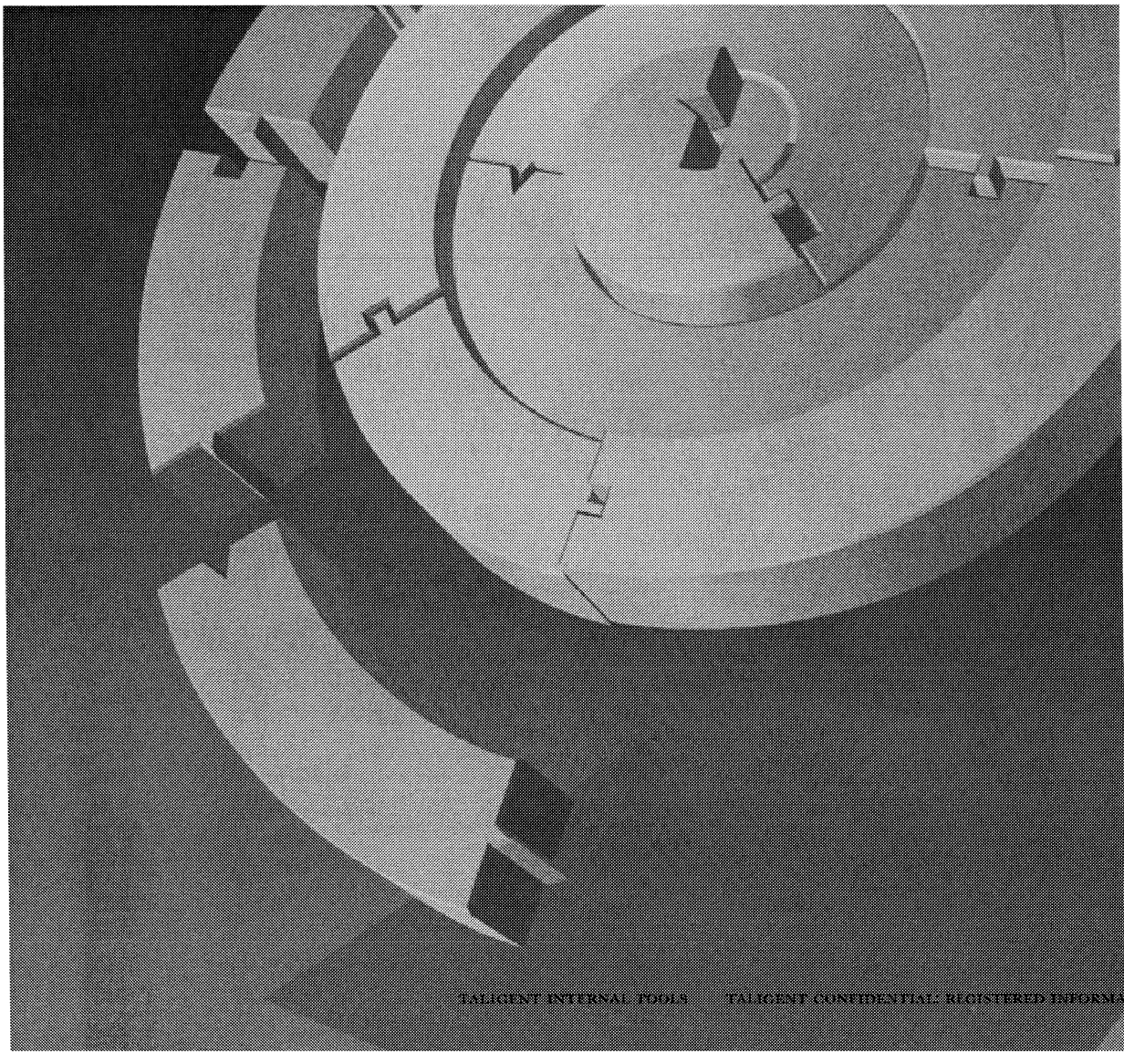
Emerge fast mode minor mode (indicator is *F*—fast mode) disables ordinary Emacs commands, and Emerge commands do not need a C-c or c-c prefix.

Emacs function keys

	Key	Function	Description
Build	F1	compile	Makeit
	Ctrl-F1	makeit-clean-complete	Makeit clean complete
	Sh-F1	makeit-c	Makeit -c
	Meta-F1	makeit-binaries	Makeit binaries
	Meta-Sh-F1	makeit-c-binaries	Makeit -c binaries
	Ctrl-Sh-F1	makeit-c-clean-complete	Makeit -c clean complete
Message and search	F2	next-error	Step to next error or search result
	F3	search-in-pinkincludes	Search for pattern in include directories
	Ctrl-F3	grep	Search for pattern
Taligent AIX layer	Ctrl-F4	start-pink-app	Start a Pink application
	Meta-Sh-F4	start-pink-xdb-app	Start a Pink application under xdb control
	Sh-F4	stop-pink	Stop Pink!

Emacs function keys

	Key	Function	Description
Got and change buffer	F5	goto-line	Go to specified line number
	Ctrl-F5	what-line	Display current line numbers
	Sh-F5	open-helpfile	Display this help file
	F6	previous-buffer	Go to previous buffer in buffer list
	F7	next-buffer	Go to next buffer in buffer list
	Sh-F6	bury-buffer	Push this buffer to the end of the buffer list
	Sh-F7	unbury-buffer	Oops--bring it back to the front
Open files	F8	open-selection	Open current selection as a file
	Ctrl-F8	cplus-comment-region	Make lines in current selection into C++ comments (insert //)
	Sh-F8	open-pink-makefile	Open the standard Pink makefile in . (current directory)
Keyboard macros	F9	start-kbd-macro	Start recording keyboard macro
	F10	end-kbd-macro	Stop recording keyboard macro
	F11	call-last-kbd-macro	Execute last recorded keyboard macro
	F12	query-replace	Query replace!
	Ctrl-F12	replace-string	Replace string!
	PrintScreen	check-out-buffer	Check out current file from SCCS
	ScrollLock	check-in-buffer	Check in current file to SCCS
	Pause	cpg-emerge	Three-way merge of file revisions
	Sh-Pause	cpg-special-merge	Three-way merge of specific builds
Navigation	Home	beginning-of-line	Move cursor to beginning of line
	End	end-of-line	Move cursor to end of line
	Ctrl-Home	beginning-of-buffer	Move cursor to beginning of buffer
	Ctrl-End	end-of-buffer	Move cursor to end of buffer
	Sh-Home	beginning-of-window	Move cursor to beginning of window
	Sh-End	end-of-window	Move cursor to end of window
	PageUp	scroll-down	Scroll down!
	PageDown	scroll-up	Scroll up!
	Ctrl-PageUp	scroll-other-window-down	Scroll other window down
	Ctrl-PageDown	scroll-other-window-up	Scroll other window up



APPENDIX B

TALIGENT SOURCE CODE MAINTENANCE

Taligent source code is stored in a hierarchy of directories maintained by our SCM tools. To check source code in and out of SCM, you must first create a directory hierarchy in your workspace that mirrors the SCM hierarchy.

To set up your working environment to work with the Taligent source code maintenance system (*SCM*), follow the steps in Chapter 2, "Working in the AIX environment."

TERMINOLOGY

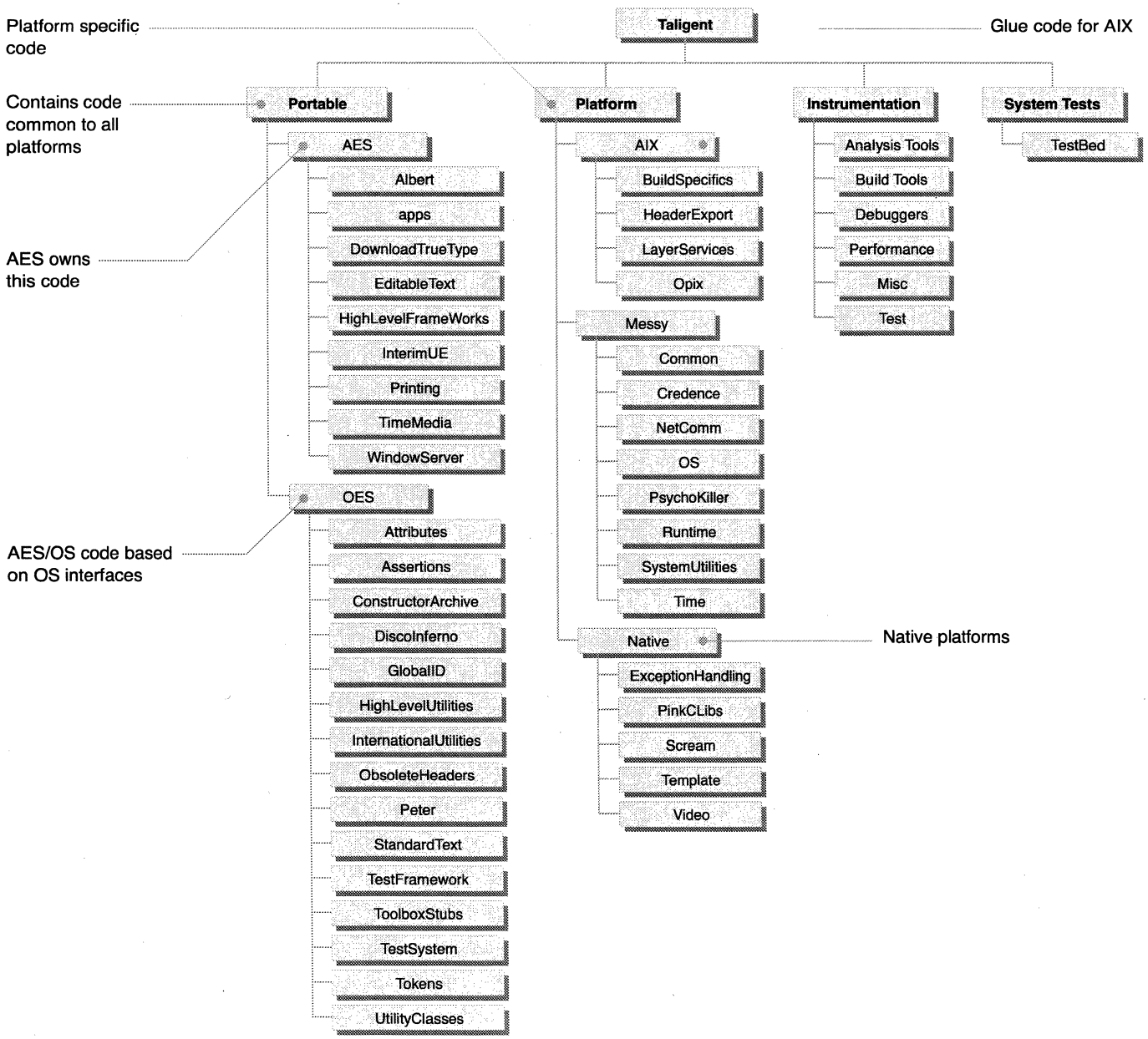
Taligent uses the following terms and definitions when discussing source code management:

- *Project*—a directory that contains source code, other projects (subprojects), or both.
- *Project hierarchy*—a tree of projects of arbitrary depth.
- *Workspace*—your own directory hierarchy that mirrors the source code project hierarchy. You check out files to your workspace.
- *TaligentRoot*—the root of your workspace hierarchy. The path to TaligentRoot is contained in the `$TaligentRoot` shell variable.
- *TaligentSCMRoot*—the root of the source-code server hierarchy. The path to TaligentSCMRoot is contained in the `$TaligentSCMRoot` shell variable.

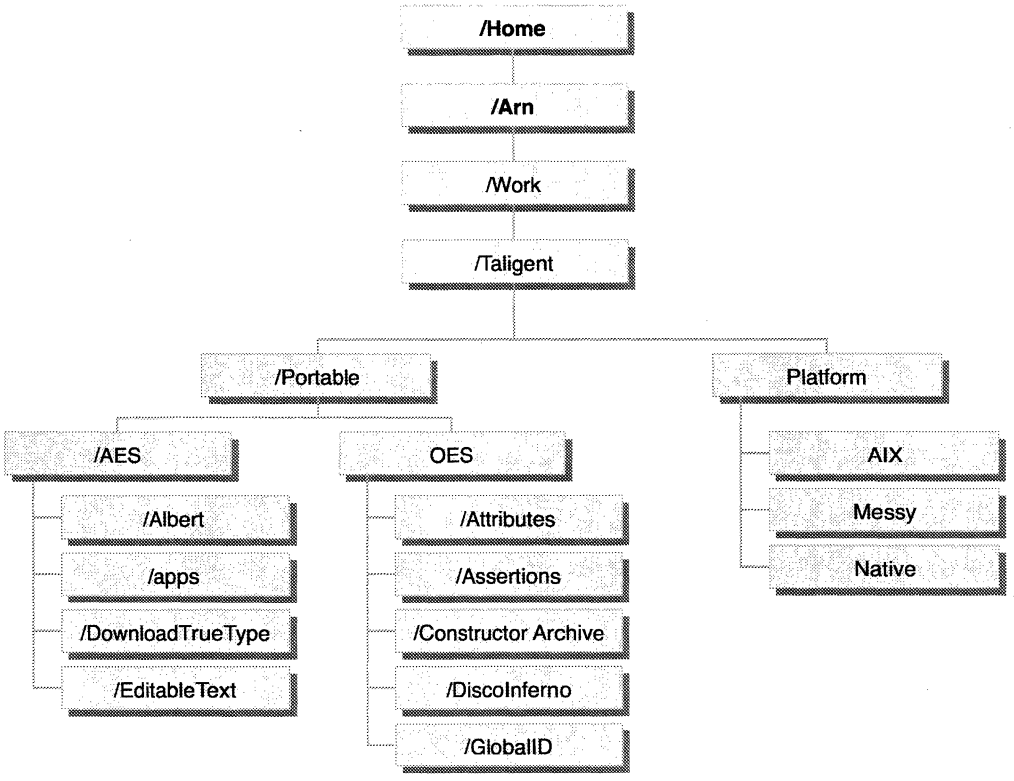
  `$TaligentSCMRoot` is a link to the SCM repository. Use this logical directory to access the repository because the physical directory can move.

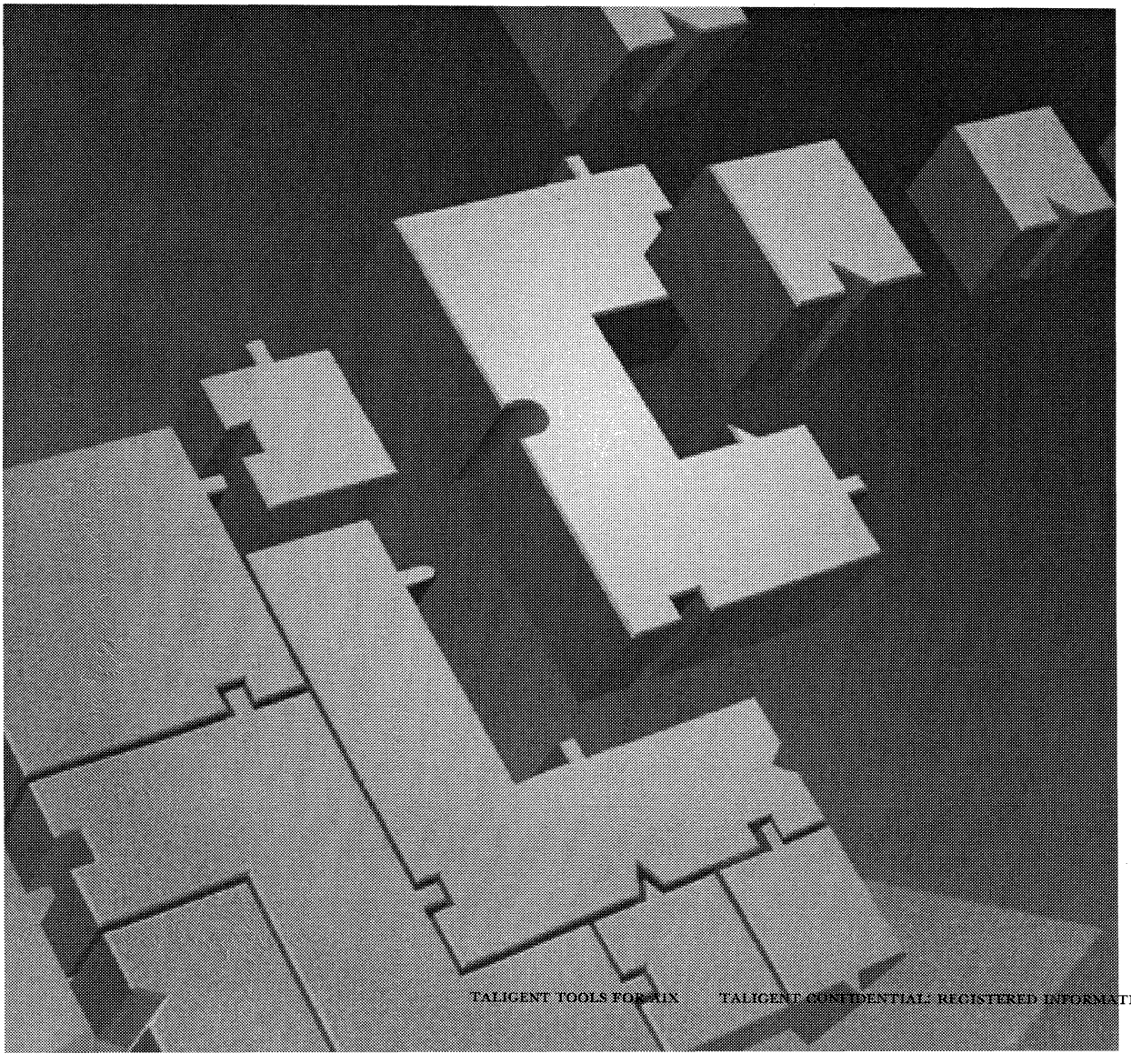
PROJECT HIERARCHY

NOTE This diagram is a snapshot of the hierarchy—the hierarchy can and will change until code freeze.



As an Taligent engineer, you have a mirror of the SCM hierarchy on your local file system. The mirrored directory structure is your *workspace* or *working directory*. When you retrieve or check out a file from the SCM hierarchy, `Checkout` places the file in your corresponding working directory. For example, if your home directory is `/home/arn`, then your working directory hierarchy is probably underneath `/home/arn/Work`.





INDEX

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X