Note: in the PDF (electronic) version of this book, the page numbers are hyperlinks. Click on any page number to view that page.
Examples: Index, p. 387, Table of Contents, p. 3.

TLIB Version Control

*** PRELIMINARY ***

Reference Manual

Printed in the U.S.A.

Version 5.53, printing 1

Table of Contents

Copyright / License

Limited Warranty

Burton Systems Software warrants for a period of ninety (90) days from the date of delivery that, under normal use and without unauthorized modification, the programs perform substantially in accordance with the specifications published in the documentation and those set forth in Burton-authorized advertising material; that, under normal use, the magnetic media upon which the programs are recorded is not defective; and that the user documentation is substantially complete and contains the information which Burton deems necessary to use the program. If, during the ninety day period, a demonstrable defect in the programs should appear, you may return the software to Burton for repair or replacement, at Burton's option. If Burton cannot repair the defect or replace the software with functionally equivalent software within thirty (30) days of Burton's receipt of the defective software, then customer shall be entitled to a full refund of the purchase price.

Burton excludes any warranty coverage for incidental or consequential damages except for the express warranties above, and limits the end-user's remedy to return of the software with manual to the dealer or to Burton for replacement.

This statement shall be construed, interpreted and governed by the laws of the State of North Carolina.

Note: Even after the expiration of the warranty, please do not hesitate to contact us about any problems, questions, or suggestions which you may have. We pride ourselves on the quality of the support which we provide to our customers, and we want you to be satisfied.

P. O. Box 4157, Cary, NC 27519-4157 USA
Telephone: (919) 481-0149   FAX: (919) 481-3787
email: **`support@burtonsys.com`**
web site: **`http://www.burtonsys.com/`**

TLIB

A Version Control System

What is TLIB?

TLIB™ (pronounced "*tee-libe*") is what is sometimes called a "version control system," "configuration management system," or a "source code librarian." It stores all versions of any of your sources file in a single, compact, annotated library file. Your source file is most likely a text file containing program source code or documentation; however, non-text "source" files (like object module libraries or spreadsheet work files) are also supported.

TLIB is especially useful if you are a programmer, since it lets you quickly and easily go back to an old version in the event that the latest version has a new bug. It can also manage many of the most vexing chores associated with the software development process, such as coordinating the modification of source code by several programmers, maintaining and reconciling parallel development paths, migrating/merging changes into customized versions, and providing an "audit trail" of revision history information.

TLIB is very fast. For example, on a 100 MHz Pentium, it will Update a library file, storing the 200th version of a 750K source file (this reference manual, as it happens) in about 4 seconds. Sluggish tools can get in the way of your productivity. TLIB doesn't!

TLIB is easy to use. It includes both a user-friendly GUI Windows interface, and command-line versions for several operating systems..

TLIB grows with your business. It works fine for small projects on standalone PCs, but it also supports shared libraries over a Local or Wide-Area Network, and multi-stage "promote" hierarchies, for big projects.

Requirements

TLIB™ Version Control includes executables to run on Microsoft Windows™ 95/98/Me/NT/2K/XP (32-bit), Windows 3.1x (16-bit), MS-DOS™ and PC-DOS™ versions 3.0 and above, and IBM™ OS/2 versions 2.x and Warp. For MS-DOS and PC-DOS, both real-mode and DOS-extended versions are included. The 32-bit command-line version also runs under the WINE Windows Emulator on Linux (Intel-architecture CPUs only), though it currently assumes case-insensitive file names.

The real-mode TLIB executable, `TLIBDOS.EXE`, requires at least 500K of available "conventional" (non-extended) RAM memory. The DOS-extended TLIB executable, `TLIBX.EXE`, requires less conventional memory, but also needs at least 1 MB of extended memory and a DPMI, VCPI or XMS memory manager, such as Windows™, `EMM386.EXE`, or `HIMEM.SYS`.

An 80286 or better CPU is also required for DOS versions of TLIB; other versions of TLIB require an 80386 or better CPU and a hard disk drive. A CD-ROM drive is required for installation (or TLIB can be purchased on 1.44MB 3.5" diskettes, by special order).

There is almost no limit on the size of source and library files. However, performance degrades for source files larger than a few megabytes.

For networked development, in which two or more users will share access to the files, either multiple copies of TLIB or a multi-user license is required. TLIB uses only basic network file support, such as file and record-locking, so it works with almost all networks. (You can use the included TESTLOCK tool to test your network's file/region sharing/locking.)

Introduction

If you purchased the TLIB Version Control on CD-ROM, then install it by inserting the CD and (if `SETUP` does not run automatically) run `SETUP.EXE` under Windows. (If you received TLIB Version Control via electronic delivery, and you are reading this, then you presumably have already installed TLIB, by downloading the files into a temporary directory and running `SETUP.EXE`.) The Windows-based installation program will install a TLIB Version Control folder shortcut containing shortcuts for `tlib_doc.pdf` (the electronic version of this Reference Manual) and other documentation files, as well as for TLIB Version Control and the TLIB Configuration Wizard.

Converting to TLIB from other products

TLIB comes with tools to automate conversion of libraries/archives/logfiles from many other products to TLIB. At this writing, we have utilities to automatically convert from SourceSafe, PVCS, MS Delta, Sorcerer's Apprentice, MKS RCS, GNU RCS, and Unix RCS.

The conversion utilities are stored in `CONVERT.ZIP`. Unpack them by running the `CONV_UNP.BAT` script. See `CONVERT.TXT` for more instructions.

TLIB Version Control Features

New Features in TLIB Version Control 5.xx:
o Easy to use Graphical User Interface, with button bar, menus, right-button functionality, helpful status-bar guidance, MRUs, etc.. Honed to smooth ease-of-use through a long beta test cycle.
o Flexible file pick-list, with multiple selection, sorting, etc..
o Full compatibility with command-line versions of TLIB and full upward-compatibility with all past versions of TLIB.
o Direct support for the compiler-native “project files” for several popular software development tools, including Visual Basic 3.0-6.0, Watcom C/C++ 10.x, Borland Delphi and C++ Builder, Symantec Visual Cafe, and Help Magician Pro. That means you can simply “open” a compiler-native project in TLIB, rather than specifying files with wild-cards and file-lists. (We will also be adding support for other compilers.)
o Fully restartable multiple-file operations. TLIB optionally deselects each file in the pick-list when done processing that file, so you can cancel the operation (or skip individual files) and later restart the command to resume where you left off or process the skipped files.
o A very nice, colorful, side-by-side visual compare, fully integrated.
o Easy to use Windows-based installation under Windows 3.1, Windows-9x/Me, Win-OS/2, and Windows-NT/2K/XP. Includes an uninstaller, too, but we doubt you'll ever use it.
o TLIB Add-Ins for Visual Basic 4.0-6.0, and for MS Developer Studio (VC++ et al) 5.0.
o Three public APIs, for integrating TLIB with your application.
o Includes command-line versions of TLIB with support for long file-names on operating systems which support them, as well as real-mode DOS and DOS-extended versions.
o Unrivaled configurability. TLIB now supports over 100 different configuration parameters. TLIB's configuration file supports “if/endif,” “include,” conditional loading, environment variable references, and full expression evaluation, including parentheses and 28 different operators.
o TLIB Configuration Wizard helps you quickly configure TLIB the way you need it.
o Very flexible wild-card specifications, including support for file-lists, multiple asterisks in wild-card specs (even under DOS or Windows 3.1x), six different wild-card search modes (most of which can be combined), and optional automatic spanning of subdirectories.

o Automatic translation of DOS, Unix, and Macintosh ASCII text files; that is, text files with all three common kinds of end-of-line delimiters: LF, CR, and CR+LF. Configurable control over which text format is generated by TLIB when extracting (“checking out”) text files. (Of course, for binary files no translation is ever done.)

Plus all the advantages of TLIB 5.00:
o TLIB Version Control supports all languages
o TLIB runs very, very fast. Most magazine reviewers have found that TLIB is noticeably faster than any other version control system
o A single library file stores all versions of a source file, with date/time, user id, and comments for each version
o Coordinates access by multiple programmers
o N-way Branching and named project-level support, for parallel development
o Branch/level-local locking option
o TLIB's unique append-in-place forward delta system provides higher reliability and recoverability than any reverse-delta product
o Open architecture: there are no secrets about TLIB's file formats, nor about where your source code is stored
o Unique Whole-Level Change Migration, eases merging of changes into customized variants of your software, or from bug-fix/release levels into development levels, etc.. If you have to manage lots of customized versions of one program, this feature is absolutely indispensable... and (as far as we know) only TLIB has it
o Full, delta-based binary file support, with true adaptive deltas
o Very flexible embedded-keyword support
o Simple snapshot-based version labeling
o Automated conversion from SourceSafe, PVCS, MS Delta, Sorcerer's Apprentice, and any of several variants of RCS
o Free tech support by phone and email. TLIB Version Control is the sole product of a small company, so there's no wading through tedious phone menus and “tech support” people who have no idea what you are talking about
o Automatic “delta” generation - only changes are stored from version to version
o Coordinated control of multiple modules: “fixed” snapshot version labels, and “floating” tracked versions for each named project level
o Supports trees of subdirectories
o Can merge (reconcile) simultaneous changes, and flag conflicts, or undo intermediate revisions without losing later changes
o “Promote” between project levels, for ISO 9001-style staged development on large projects
o Optional reference directories for each level

o Revision history documents changes, and central activity journal for “audit trail” of development activity, including revision comments, for entire project
o Create file-lists by scanning source code for includes (C, Pascal, MASM, QuickBasic, COBOL “copy”, many others)
o Highly flexible user-formatted keyword support
o Keyword-based version number verification, warns if you store an obsolete version, even if you disable check-in/out locking
o Efficient support of local and wide area networks, remote access nodes, WORM optical drives
o Generate mainframe-compatible deltas in any of four formats
o Integrated with Opus Make and most good programmers' editors
o DOS-extended version included (more efficient for very large files)
o Automated conversion of archives from other version control systems
o ISO 9001-style “promote” structures
o Supports multiple, named project levels, including customization levels
o Both sparse and fully-populated project levels, and conversion between them
o Automatic reference directory refresh, per project level
o The “EBF” fast-extract command, refreshes browse-mode files
o N-way branching
o Automatic version tracking and automatic branching
o Branch/level locking and weak (warning-only) locking
o Support for trees of subdirectories
o Comma-delimited in-line file lists
o Improved command structure, with command synonyms
o Keyword-based version number verification
o Environment variable substitutions in `TLIB.CFG`
o Autoset file, for “local” environment variables
o Over 100 configuration options, for customizing to your taste

TLIB Library Files

TLIB creates a library file for each of your source files. The library file can contain every version of the source file which ever existed, so you'll never again need to worry about whether you can safely erase an old source file. However, only the changes from one version to the next are actually stored in the library file, so it remains modest in size even when it contains dozens or hundreds of versions.

The name of a library file is normally the same as that of the corresponding source file, except for the extension. The way that TLIB determines the extension is user-configurable, but with the most common settings a library file has the same extension as the corresponding source file except for the second character, for which a dollar sign is substituted. Thus, for example, the library file for `XXX.PAS` is named “`XXX.P$S`” (probably in a different directory). This is the convention we'll use for most of the examples in this manual. However, other conventions can also be used; see p. 262.

Whenever you update a library file from a source file, the library file is appended with a *delta* or `version definition' of ‘edit commands.’ These edit commands record what changes were made to the source file since the previous version: the lines (or binary data) which were added, deleted, and moved. The library file contains one delta for each version of your source file. Since most of the source file is usually unchanged, the library grows in size only a little for each update. Yet you can still retrieve any version from the library in just seconds.

Getting started: the TLIB Configuration Wizard

Here's a hint for using the TLIB Reference Manual: **When in doubt, consult the index.**

After you have installed TLIB by running `setup.exe`, you need to create a simple TLIB configuration file.

A TLIB configuration file is just a small ASCII text file, usually called `tlib.cfg`, which TLIB reads when it starts up. It contains “configuration parameters” to customize TLIB's behavior to your needs.

There are over one hundred different configuration parameters which you can specify. However, most users need only a few of them, at least when getting started.

To help you configure many of the most commonly needed parameters, TLIB now comes with a configuration set-up program we call the “Configuration Wizard” (which replaces an older program called `TLIBCONF.EXE`). Both the Configuration Wizard and TLIBCONF work by asking you some questions and then building an appropriate TLIB configuration file.

Even if you need to do strange and unique things, you still should begin the process of configuring TLIB by running the Configuration Wizard. Then edit the resulting `tlib.cfg` file to add your customizations.

TLIB cannot be used without a configuration file. The Configuration Wizard (or TLIBCONF) will create a starting configuration file. For simple development environments, that might be all you need to do to configure TLIB. However, for complex dvelopment environments, the `tlib.cfg` file created by the configuration wizard is only a starting point. You'll still probably need to make some manual additions to it.

Note: the rest of this chapter is out-of-date. It describes the obsolete TLIBCONF program, which has been replaced by the TLIB configuration Wizard.

Here are some examples of the kinds of questions that TLIBCONF asks:

1. LOCKING: Which best describes you?

A) A single programmer working alone on a project. You do not need check-in/out locking.
(`locking N`, `loguser N`)
B) One of a group of programmers working on the same project (or the System Librarian for a group of programmers) using a network (LAN) and multiple copies of TLIB. You need check-in/out locking. This choice also causes “browse-mode” source files to be set to read-only, so as to distinguish them from source files which are checked-out for modification.
(`locking Y`, `readonlyb Y`, `replrobr Y`, `loguser Y`)
etc...

2. FILETYPE & TABS: Which best describes the source files that you need to manage?
A) Plain ASCII text files, with no tab characters... (`entabu Y`)
B) Plain ASCII text files, but with tabs... (TLIB must not do automatic tab/blank conversions.) (`entabu N`)
C) Non-ASCII files, files containing binary data... (`filetype binary`)
etc...

3. BRANCHING: Do you often have more than one “latest” version...

4. SOURCE FILE NAMES: Might you ever need to have TLIB manage two or more files with the same names but different extensions...

5. COMMAND & PROMPT STYLES: Do you want TLIB 5.50 to mimic earlier versions of TLIB; would you like a verbose prompt or a terse one...

6. TREEDIRS: Do you keep all of your source files in a single work directory, or do you use a “tree” of subdirectories, grouping your source files by purpose into the various subdirectories...

Note: TLIBCONF and the Configuration Wizard configure rather plain prompt and help screens for the command-line versions of TLIB. If you'd like a prettier prompt/menu and help screen, see `ANSIFY.AWK`, p. 333 (or, better yet, use the Windows version of TLIB).

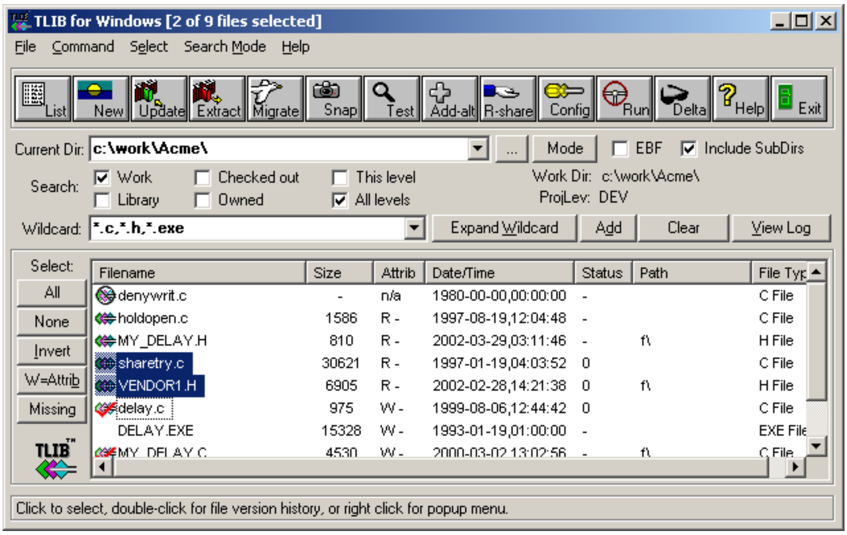
Command structure

Command-line versions of TLIB use commands which consist of a single command character, with optional suffix characters to modify the command's behavior or scope. (You can also use this form of TLIB command in the GUI *TLIB for Windows*, via “Run Manual Command” on the “File” menu, or via the “Run” button. However, it is generally more convenient to use the menus and/or buttons.)

For example, when **U**pdating libraries, you can choose between **F**ast or regular update, between **M**inor-version-number-incremented or regular (major) number incremented, either checking-in (unlocking) or **K**eeping checked-out, etc. Thus, “UFM” (or, equivalently, “UMF”) means **U**pdate with **F**ast mode, and increment the **M**inor version number instead of the main integer version number.

For more information on the TLIB 5.5x command structure, see pp. 59 and 330.

GUI (Graphical User Interface) versions of TLIB support the same commands and options, but they are specified by buttons, menu choices, and/or check-boxes. This is what the main TLIB GUI window looks like:


TLIB for Windows [2 of 9 files selected]
File Command Select Search Mode Help
List New Update Extract Migrate Snap Test Add-alt R-share Config Run Delta Help Exit
... Mode
Expand Wildcard Add Clear View Log
Select: All None Invert W=Attrib Missing
TLIB

Current Dir: c:\work\Acme\ ☐ EBF ☑ Include SubDirs

Search: ☑ Work ☐ Checked out ☐ This level
☐ Library ☐ Owned ☑ All levels

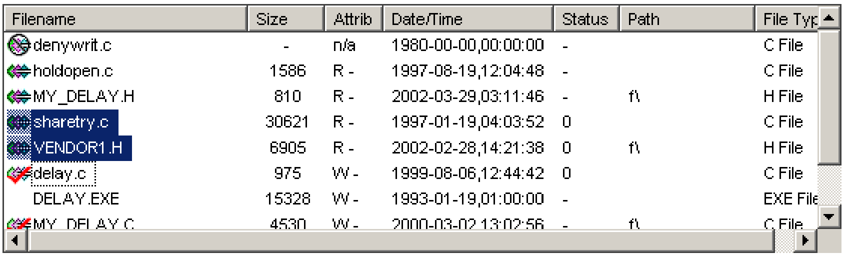
Work Dir: c:\work\Acme\
ProjLev: DEV

Wildcard: *.c,*.h,*.exe

Filename	Size	Attrib	Date/Time	Status	Path	File Typ
denywrit.c	-	n/a	1980-00-00,00:00:00	-		C File
holdopen.c	1586	R -	1997-08-19,12:04:48	-		C File
MY_DELAY.H	810	R -	2002-03-29,03:11:46	-	f\	H File
sharetry.c	30621	R -	1997-01-19,04:03:52	0		C File
VENDOR1.H	6905	R -	2002-02-28,14:21:38	0	f\	H File
delay.c	975	W -	1999-08-06,12:44:42	0		C File
DELAY.EXE	15328	W -	1993-01-19,01:00:00	-		EXE File
MY_DELAY.C	4530	W -	2000-03-02,13:02:56	-	f\	C File

Click to select, double-click for file version history, or right click for popup menu.

The most prominent feature is a big window listing your source files. They may be all in one main work directory, or they may be in a “tree” of directories, with the subdirectory names shown in the “path” column. In the pictured example, most of the files are in the main work directory, but three of them are in the “`f`” subdirectory:



Filename	Size	Attrib	Date/Time	Status	Path	File Typ
denywrit.c	-	n/a	1980-00-00,00:00:00	-		C File
holdopen.c	1586	R -	1997-08-19,12:04:48	-		C File
MY_DELAY.H	810	R -	2002-03-29,03:11:46	-	f\	H File
sharetry.c	30621	R -	1997-01-19,04:03:52	0		C File
VENDOR1.H	6905	R -	2002-02-28,14:21:38	0	f\	H File
delay.c	975	W -	1999-08-06,12:44:42	0		C File
DELAY.EXE	15328	W -	1993-01-19,01:00:00	-		EXE File
MY_DELAY.C	4530	W -	2000-03-02 13:02:56	-	f\	C File

In the Win32 TLIB GUI, each source file listed has a small icon beside the file name, which tells you something about it:

A tiny TLIB icon is shown for source files that are under version control (i.e., for which a corresponding TLIB library file exists).

A red check-mark is superimposed for files that you have checked out for modification.

A red backslash is superimposed for files that someone else has checked out for modification (locked), so that if you were to try to Extract/check-out for modification it would fail. (A blue backslash is similar, but you can still check-out the file for modification; this is only possible if “weak” or “branch/project-level” locking is being used instead of “full” locking -- see the "LOCKING" configuration parameter.)

A black circle and slash means that the source file is missing from your work directory.

These icons may be combined in various ways, too. For example, a black circle and slash superimposed over a TLIB icon means that the source file is missing from your work directory, but the corresponding TLIB library file exists (so you can extract the missing file from TLIB if you need it).

The “Filename” column shows your source file names (including binary “source” files). You can click on the “Filename” header to alphabetize the list, or click it twice for reverse-alphabetical order.

The “Size” column shows the size in bytes for each source file (or “-” if the source file is missing from your work directory).

The “Attrib” column shows the DOS/Windows file attributes (“R” for Read-only, “W” for Writable, “A” for Archive (modified), or “n/a” if the source file is missing from your work directory. If check-in/out locking is enabled (with the usual TLIB configuration settings), then the source files that you've Extracted/checked-out for modification will have the “W” (writable) attribute but “browse mode” files will be read-only. By clicking on the “Attrib” column header, you can sort the list of source files to group together those source files with the same attributes (e.g., all the files that you have edited, because they have the “A” attribute).

The other columns are self-explanatory, except for “Status.” Before you attempt to do any TLIB commands, it shows “-” for each file. After you do a TLIB command, it shows a “completion code” or errorlevel for each file that you attempted the command upon. “0” means success, and anything else (usually “1”) indicates an error.

TLIB Command Summary

command		*page*
? or **?0**	Display “help” screen	
A or **A0**	Add files to the current project level	168
AD	with Delete suffix: delete from current project level	169
AF	with Fast suffix: populate a sparse project level	171
AP	with Promote suffix: add to promote (“p=”) level	170
AS	with Specify-version suffix: specify particular version	169
AX	with eXclude suffix: mark files as excluded, with v=x	169
ADF	depopulate project level (make sparse)	172
APX	mark eXcluded file as eXcluded in the promote level	170
C or **C0**	Change any configuration parameter	252
CP	with Path suffix: configure library path	42
CW	with Who-are-you suffix: configure user id	100
E or **E0**	Extract file from library	35
EB	with Browse-mode suffix: do not lock	100
EBF	with Fast/Freshen suffix, to refresh browse-mode files	163
ER	with Reserve suffix: just lock, don't actually extract	101
ES	with Specify-version suffix: specify particular version	37
EBS, **EBFS** combinations		
F or **F0**	Filter file names with wild-card specs	64
L or **L0**	List versions	41
M or **M0**	Migrate changes	178
MF	with Fast suffix, to quietly skip already-migrated files	178
MS, **MSS**	with Specify-version suffix(es): specify “to” and/or “base”	185
MFS, **MFSS** with both Fast and Specify-version suffix(es)		
N or **N0**	New library create	24
NF	with Fast suffix, to quietly skip existing libraries	31
NK	with Keep-checked-out/check-out suffix, to keep locked	101
NS	with Specify-version suffix: specify starting version	239
NFK, **NFS**, **NFKS** combinations		
Q or **Q0**	Quit	
R	Reclassify/share-file	
S or **S0**	Create a “snapshot” version label	92
SS	with Specify-version suffix: specify particular version	92
T or **T0**	Test check-in/out lock status	102
U or **U0**	Update library with new version, check-in/unlock	32
UB	with create-Branch suffix: make a “.1” version	70
UD	with Discard-change suffix: just break lock, no update	101
UF	with Fast/Freshen suffix: only update newer files	40

command		*page*
UK	with Keep-checked-out suffix: update and keep locked	101
UM	with Minor suffix: update & increment minor version	74
US	with Specify-version suffix: specify particular version	69

UBK, **UBF**, **UBFK**, **UKM**, **UKS**, **UFK**, **UFM**, **UFKM** combinations

Alphabetical Command & Suffix Summary

Command	Suffixes allowed	Purpose	Page
`A`	`D,F,P,S,U,X,`*wc*`[A/T]`	Add/Alter project level	168
`C`	`P,W`	Configure	252
`E`	`B,F,R,S,`*wc*`[L]`	Extract source file	35
`F`		Filter files	64
`L`	*wc*`[L]`	List versions	41
`M`	`F,S,`*wc*`[L]`	Migrate changes	178
`N`	`F,K,S,`*wc*`[W]`	New library	24
`Q`		Quit	
`S`	`S,`*wc*`[L]`	Snapshot version label	92
`T`	*wc*`[C]`	Test lock status	102
`U`	`D,F,K,M,S,`*wc*`[W]`	Update library	32
`?`		Help	

Note #1: The order in which suffix characters are specified does not matter (except in the `COMMANDS` configuration parameter). Thus, for example, "**EBS**" and "**ESB**" are equivalent (both mean **E**xtract **S**pecified version in **B**rowse mode).

Note #2: "*wc*`[x]`" means wild-card search-mode suffixes are allowed, and that the default wild-card search-mode suffix is "`x`". The possible wild-card search modes are: `A`, `C`, `L`, `O`, `T`, `W`; also `I` (include subdirectories), and `N` (no wild-card expansion). (Wild-card search modes affect the way that TLIB interprets wild-card file specifications; see p. 59)

Alphabetical Suffix Summary

Suffix	Type	Commands	Meaning
`0`	null	`all`	does nothing
`A`	wildcard	`A‡,E,L,N,S,T,U,M`	search **A**ll levels
`B`	regular	`E`	**B**rowse mode (don't lock)
`C`	wildcard	`A,E,L,N,S,T†,U,M`	search all **C**hecked-out files
`D`	regular	`A`	**D**elete from project level
`D`	regular	`U`	**D**iscard changes (unlock)
`F`	regular	`A,E,N,U`	**F**ast/freshen
`I`	wildcard	`A,E,L,N,S,T,U,M`	**I**nclude subdirectories
`K`	regular	`N,U`	**K**eep checked-out/locked
`L`	wildcard	`A,E†,L†,N,S†,T,U,M†`	search **L**ibrary files
`M`	regular	`U`	**M**inor version no. increment
`N`	wildcard	`A,E,L,N,S,T,U,M`	**N**o wild-card expansion
`O`	wildcard	`A,E,L,N,S,T,U,M`	search **O**wn checked-out files
`P`	regular	`A`	**P**romote
`P`	regular	`C`	**P**ath of library/lock files
`R`	regular	`E`	**R**eserve (just lock, no extract)
`S`	regular	`A,E,N,U,M`	**S**pecify version number
`T`	wildcard	`A‡,E,L,N,S,T,U,M`	search **T**his project level
`U`	regular	`A`	**U**ndo (unimplemented)
`W`	regular	`C`	**W**ho are you (ID)
`W`	wildcard	`A,E,L,N†,S,T,U†,M`	search regular **W**ork files
`X`	regular	`A`	e**X**clude from project level

† indicates that this wild-card search mode suffix is the default for the indicated command.

‡ indicates that whether or not this wild-card search mode suffix is the default for the indicated command depends upon which regular suffixes are used.

N Command: create New library

The command-line version of the main TLIB program is `TLIB.EXE`. It is actually a copy of one of several command-line versions: `TLIB32C.EXE` (for Win32), `TLIBDOS.EXE` (for real-mode DOS), `TLIBX.EXE` (for DOS-extended protected-mode, or `TLIB2.EXE` (for OS/2). To run it in interactive mode, type the program name, e.g.:

```
 TLIB
```

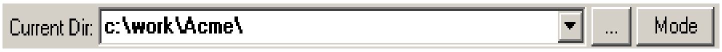
The GUI versions of TLIB are called `WTLIB32.EXE` (Win32) and `WTLIB16.EXE` (16-bit Windows, a/k/a Win 3.1x & Win-OS/2), but you don't need to remember that because they are usually started via Start Programs menu shortcuts.

Note: The examples in this reference manual all show the Win32 TLIB GUI, but the 16-bit GUI is similar.

In the TLIB Windows GUI, commands are performed by first listing and selecting the files to be operated upon, then picking the command and options. Command-line versions of TLIB require the opposite order of events: first you specify the command, then you specify the files (typically via wild-cards and/or file lists).

For both GUI and command-line versions of TLIB, the first thing you must do to run it (after installing and configuring it) is to select the correct “current” work directory. In command-line TLIBs, this is done with the “`cd`” (change directory) DOS command.

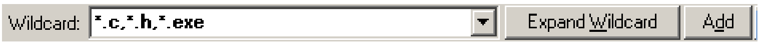
In the TLIB GUI, select the current directory via the “Current Dir” frame:



You can click on the “...” button to browse for the desired directory, or select a recently used directory by clicking on the ▾ MRU-dropdown button, or simply enter the desired directory in the text entry box.

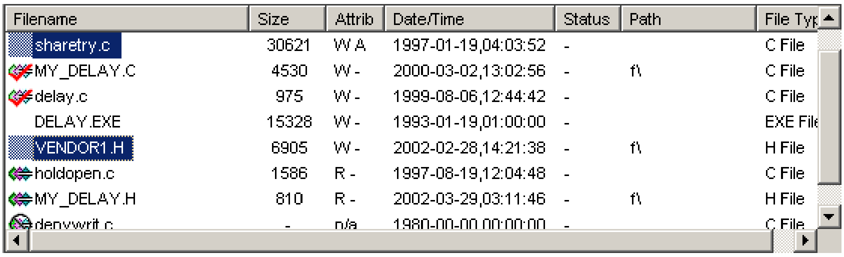
(Or, if you click the “Mode” button to switch TLIB into “native project mode,” the current directory will be implied by the location of the “native” compiler or editor project file which you select.)

In the TLIB GUI, the next step is to identify your source files to TLIB. In native project mode, TLIB deduces them by reading a “project file” that was created by your compiler or editor (or a file list or HTML index file). In “regular” (file-oriented) mode, you specify your source files to TLIB via one or more wildcards and/or “`@file.lis`” filelists:



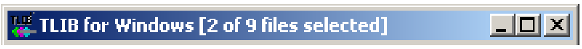
Simply enter the wildcard specifications, separated by commas, and then press Enter or click “Expand Wildcard.” (The “Add” button is similar to “Expand Wildcard” except that it adds more files to the existing list of foles.)

Next, select the files to be stored into TLIB, using your mouse and/or the “Select” buttons on the left side of the screen:

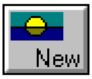


Filename	Size	Attrib	Date/Time	Status	Path	File Typ
sharetry.c	30621	W A	1997-01-19,04:03:52	-		C File
MY_DELAY.C	4530	W -	2000-03-02,13:02:56	-	f:\	C File
delay.c	975	W -	1999-08-06,12:44:42	-		C File
DELAY.EXE	15328	W -	1993-01-19,01:00:00	-		EXE File
VENDOR1.H	6905	W -	2002-02-28,14:21:38	-	f:\	H File
holdopen.c	1586	R -	1997-08-19,12:04:48	-		C File
MY_DELAY.H	810	R -	2002-03-29,03:11:46	-	f:\	H File
denywrit.c	-	n/a	1980-00-00,00:00:00	-		C File

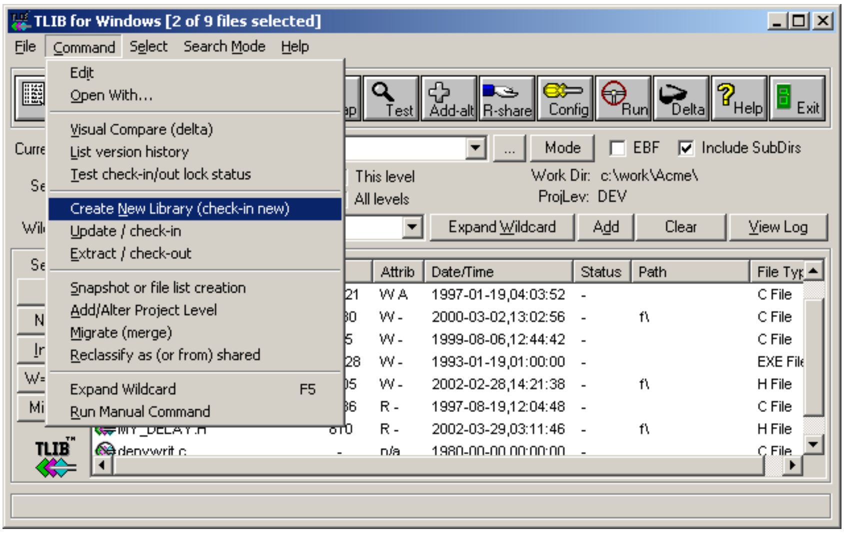
The title bar at the top of the window will change to indicate how many files you have selected:



Finally, you are ready to pick the “New libraries” command, to place your selected source files under version control, storing the first version of each file into a TLIB library file. You can find the command two ways:

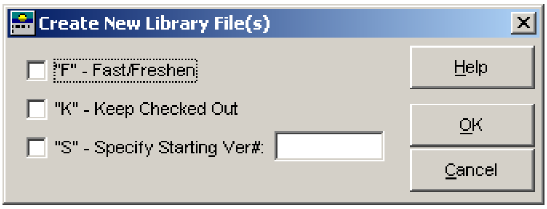
1) The “New” menu button: 

2) “Create New Library” in the command menu, which you can view either by clicking “Command” in the top menu bar, or by typing alt-C, or by right-clicking anywhere in the main file list:



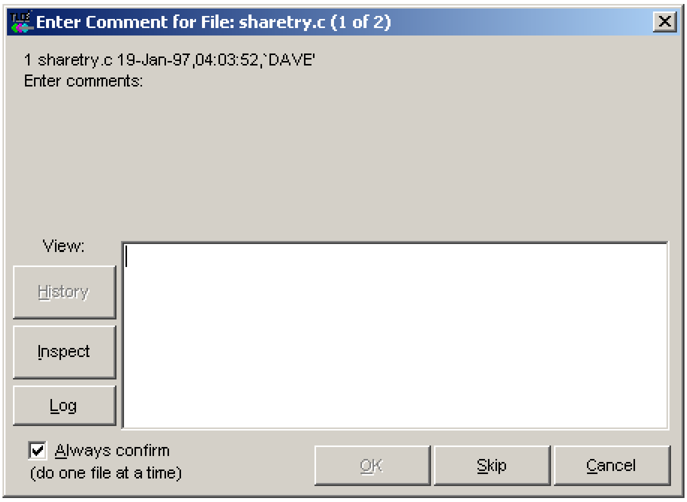
```
21  W A  1997-01-19,04:03:52  -        C File
30  W -  2000-03-02,13:02:56  -  f\    C File
5   W -  1999-08-06,12:44:42  -        C File
28  W -  1993-01-19,01:00:00  -        EXE File
05  W -  2002-02-28,14:21:38  -  f\    H File
36  R -  1997-08-19,12:04:48  -        C File
MY_DELAY.H  810  R -  2002-03-29,03:11:46  -  f\    H File
denywrit.c  -    n/a  1980-00-00,00:00:00  -       C File
```

When you pick the “New” command, you'll then see an options screen, like this:



Note: Hover the mouse cursor over any check-box to see a longer description of its purpose.

Then click “OK” to start storing your source files into TLIB, and TLIB will give you an opportunity to enter comments about each source file:

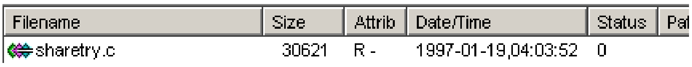


1 sharetry.c 19-Jan-97,04:03:52,`DAVE'
Enter comments:

The "OK" button is disabled (greyed-out) until you enter a comment.

To store multiple files with the same comment, uncheck the "Always confirm" checkbox.

After you've completed the "New libraries" command, and stored your source files into TLIB, the main file list will change to show updated icons, "Attrib" and "Status" for each file:



Filename	Size	Attrib	Date/Time	Status	Pat
sharetry.c	30621	R -	1997-01-19,04:03:52	0	

Command-line TLIB

When you run a command-line version of TLIB in interactive mode (i.e., without parameters), you will be presented with a prompt or menu. The appearance depends on how TLIB is configured, but one possibility looks something like this:

```
╔════════════════════════╗
║ TLIB Version Control   ╠══════════════════════════╗
╠════════════════════════╝ U=Update library          ║
║ E=Extract source       │ UM=increment Minor#       ║
║ EB=for Browse          │ US=Specify branch#        ║
║ ES=Specify version#    │ UK=Keep locked            ║
║ ER=Reserve/just-lock   │ UD=Discard/no-update      ║
║ EBS=Browse+Specify#    │ UKM,UKS=combinations      ║
║────────────────────────────────────────────────────║
║ N,NF=New library │ L=List  │ T=Test   │ Q=quit     ║
║────────────────────────────────────────────────────║
║ A=Add-to/Alter projlev   AP=Promote   │ ?=help     ║
║────────────────────────────────────────────────────║
║ C=Configure   CW=Who   CP=Path of library  ╔═══════╝
╚════════════════════════════════════════════╩═>
```

Note that this prompt might *not* give a complete list of TLIB's commands. Instead, it shows those commands which you are likely to need, based on your answers to the questions asked by the Configuration Wizard or TLIB-CONF. Another possible prompt is simply:

```
TLIB command (? for help):
```

Note: If you don't like the prompt which the Configuration Wizard or TLIBCONF configured for your use, you can easily customize the prompt (and other aspects of command-line TLIB's user interface). See the `PROMPT`, `HELP`, `BANNER` and `COMMANDS` configuration parameters, pp. 329 & 330.

To create a new library file for an existing source file, type "`N`", for "new library." You will be prompted for the name of the source file:

```
Create library from what source file?
```

You should type the name of the source file which you want saved into a TLIB library file, then press `ENTER` (or `RETURN` on some keyboards). Multiple files can be specified with wild-cards or file lists; see p. 50.

If you entered `XXX.PAS` (for a Pascal program named `XXX`), TLIB will attempt to read the file and then create a library file called (perhaps) `XXX.P$S`. If `XXX.PAS` does not exist, or if `XXX.P$S` already exists, then TLIB will display an error message and return to the main menu. Otherwise, TLIB will prompt you for a comment:

```
Comment line?
```

The comment line is your description of the source file, which you will use to identify this version if you ever need to retrieve it from the library file. For example, you might enter a comment like this:

```
 A program to count widgets
```

If you press `ENTER` without typing a comment line, TLIB will return to the main menu without creating the library file. If you want to enter a blank comment line (not good practice!), you can type a space before pressing `ENTER`.

If your comment will not fit on a single line, end the first line with a backslash (\) character, and TLIB will prompt you for another comment line. You can repeat this with successive comment lines to enter comments of any length. (Note: If you frequently enter multi-line comments, you may wish to configure TLIB with the “`SlashCont M`” parameter, so that you will not have to enter backslashes when typing multi-line comments; see p. 285.)

If, after entering one or more comment lines, you change your mind, you can abort TLIB by pressing Ctrl-Break.

After you enter your comment line(s), the library file (e.g., `XXX.P$S`) will be created with your comments and with `XXX.PAS`'s creation date associated with the first version. (Note: other information, such as the date/time, user ID, etc., is also usually stored in the library file; see `loguser`, p. 266.)

MS-DOS users: We recommend that you use CED, RETRIEVE, DOSEDIT, or a similar utility to enhance DOS's line editing. These work better than DOS's inferior DOSKEY utility, since DOSKEY doesn't keep a separate command history for applications (though DOSKEY is better than nothing). It will be helpful for both TLIB comment lines and DOS command lines. (However, CED has compatibility problems with the DOS box under some versions of Windows.) See p. 365.

Note: you needn't use the N command at all if you configure `UPDATENEW Y`, since TLIB will then create a missing library file automatically, when you try to to store the first version of your source file with the U (update) command.

Avoiding the prompt

Important note: To avoid being prompted for a comment for each of your files, you can run TLIB with command-line parameters, like this:

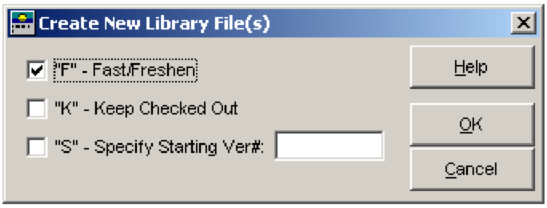
```
 TLIB N *.C,*.H short comment
```

This example creates new libraries for all your `.C` and `.H` files, each with the comment, “`short comment`”, and it won't prompt you for comments for each file (unless `SLASHCONT N` is configured, see p. 285).

NF command: Fast New library create

TLIB also supports the F (fast/freshen) suffix on the N command. The only difference between the NF command and the regular N command is that the NF command will not complain about already-existing library files; it just silently skips them.

In the TLIB GUI, you can select F (fast/freshen) via a checkbox:



[✓] "F" - Fast/Freshen
[ ] "K" - Keep Checked Out
[ ] "S" - Specify Starting Ver#:

The NF command makes it more convenient to use wild-cards to create library files for new source files (especially when you don't configure `UPDATENEW Y`). For example, a dBase programmer who had created some new source files could type one of the following commands:

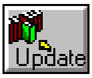
```
 TLIB NF *.PRG
 TLIB NF *.PRG Modules added in rel2
```

Both commands will create the TLIB library files for all .PRG files for which the library files did not already exist. The first example will cause TLIB to prompt for the comment for each file. The second example will just use “Modules added in rel2” as the comment, and won't prompt (unless `SLASHCONT N` is configured, see p. 285).

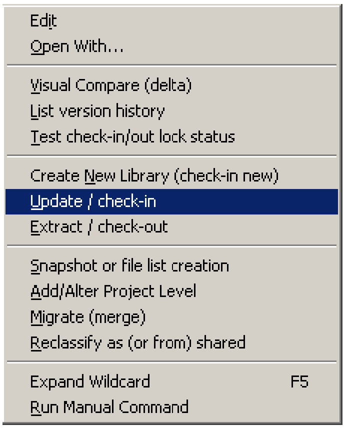
U command: Update a library

After you make a change to your source file, you can update the TLIB library and store ("check-in") the new version by selecting "Update" in the TLIB GUI, or by running command-line TLIB and pressing U (for Update).

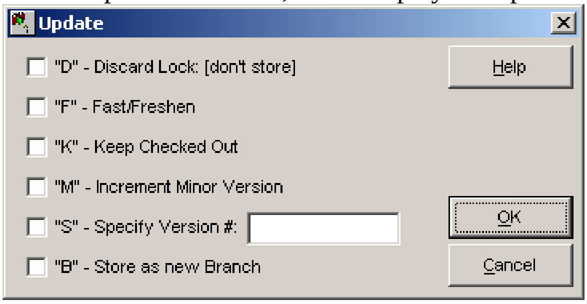
If using the TLIB GUI, first select the file(s) that you want to store, then pick the "Update / check-in" command. You can find the command two ways:

1) The "Update" menu button: 

2) "Update / check-in" in the command menu, which you can view either by clicking "Command" in the top menu bar, or by typing alt-C, or by right-clicking anywhere in the main file list:



After you pick the Update command, TLIB displays an options screen:



- [ ] "D" - Discard Lock: [don't store]
- [ ] "F" - Fast/Freshen
- [ ] "K" - Keep Checked Out
- [ ] "M" - Increment Minor Version
- [ ] "S" - Specify Version #:
- [ ] "B" - Store as new Branch

Note: Hover the mouse cursor over any check-box to see a longer description of its purpose.

Then click “OK” to start storing your source files into TLIB, and TLIB will give you an opportunity to enter comments about each source file:

When doing an U (update) command with a command-line version of TLIB, as with the N command you will be prompted for the file name, but this time TLIB expects both the source file and the TLIB library file to already exist (or you can configure TLIB to automatically create missing library files, see the `UpdateNew` parameter, p. 276). TLIB will prompt you:

```
  Update library for what source file?
```

You should enter the name of the source file.

TLIB will then read both library and source files. If the latest version in the library file is identical to the source file, TLIB will normally display:

```
  No changes.
```

(However, TLIB can be configured to add a new version even if there were no changes; see the `ForceU` configuration parameter, p. 275.)

Otherwise, you will be prompted for a comment to be associated with the new version:

```
  Comment line?
```

You might enter, for example:

```
  Faster version -- do I/O in big blocks.
```

or perhaps:

```
  Fix bug so it'll handle >32767 widgets
```

If you press `ENTER` without typing a comment line, TLIB will return to the main menu (or skip this file and go on to the next one) without updating the library. (Note: in general, pressing `ENTER` at any TLIB prompt will abort the operation.)

If your comment will not fit on a single line, end the first line with a backslash (\) character, and TLIB will prompt you for another comment line. You can repeat this with successive comment lines to enter comments of any length.

If, after entering one or more comment lines, you change your mind, you can abort TLIB by pressing Ctrl-Break; TLIB will exit leaving the library file unchanged. Or (unless you've configured `SlashCont M` or `SlashCont N`) you can abort the update by entering a 0-length comment line.

Otherwise, TLIB will add the new version to the library file, recording the date and your comments.

Delta Review

If you want to see the “delta” (changes) before entering your comments in command-line TLIB, enter “`?`” and TLIB will display the changes on your screen. This can help remind you of why you modified the module, so that you can enter meaningful comments. We call this feature *delta review*.

After displaying the delta, TLIB returns to the “`Comment line?`” prompt.

The delta format is described on pages 228 and 371.

Note: The Windows version of TLIB has a vastly better Visual Compare, instead of Delta Review.

Safety

The U command (and the various multi-character commands starting with U) are the only TLIB commands which modify an existing library file. They merely append the delta to the end of the library file without changing any of the information which is already there. Thus you are unlikely to lose any data, even if the electricity fails while you are doing an update. Nevertheless, we recommend that you back up your library files frequently, “just in case.”

E command: extract latest version

To retrieve a copy of the latest version of your source file from a library file, use the E (extract) command. You will be prompted for the name of the file which you wish to retrieve:

```
Extract what source file from library?
```

If you want to extract `XXX.PAS` from its TLIB library (which is most often named `XXX.P$S`), enter `XXX.PAS`. If `XXX.P$S` does not exist, TLIB displays an error message. If `XXX.PAS` already exists in the current directory, you will be asked whether to replace it with the version from the library. (However, TLIB can also be configured to silently replace already-existing files, or to abort without operator confirmation; see the `REPLACE` parameter, p. 268.)

You can do the same thing non-interactively, by putting the command and file name on the DOS command line. For example:

`TLIB E XXX.PAS` *(extracts* `xxx.pas` *into the current directory)*

`TLIB E SUBD\XXX.PAS` *(extracts* `xxx.pas` *into subdirectory* `.\subd`*)*

TLIB lets you specify as little or as much as you wish at each step. So, for example, you could run TLIB like this:

```
TLIB E
```

and TLIB would prompt you for the file name to extract.

Or, if you run TLIB in interactive mode:

```
TLIB
```

then when TLIB asks you for a command, you could type “`E XXX.PAS`” to avoid being prompted for the file name.

By default, TLIB libraries are in “text format,” which means that they are designed for storage of ASCII text files, not arbitrary “binary files.” An ASCII source file extracted from a text-format (normal) TLIB library is identical to the latest version which was saved in the library except that:

1) All lines, including the last one, will end in carriage-return/line-feed. For input files, TLIB allows lines to end in either carriage-return/line-feed or just a carriage-return alone (except for the last line in the file, which needn't have either). That is, the line-feeds are optional. However, when TLIB reconstructs the files, *all* lines will end in carriage-return/line-feed.

2) TLIB will not put a Ctrl-Z at the end of the file unless you configure `AddCtrlZ Y` (see p. 276).

In addition, if you enable TLIB's automatic tab/blank and blank/tab conversions (see `entabU` and `detabE`, p. 256), then:

3) Tabs will have been converted to blanks.

4) Any lines with trailing blanks (blanks immediately preceding the carriage-return) will have the trailing blanks removed.

5) Lines of more than 254 characters long may be truncated or split. TLIB cannot do blank/tab conversions in files with extremely long lines.

It is also possible to store “binary” source files in TLIB libraries. With binary format libraries, TLIB will never make *any* changes to your files. Use the `FileType Binary` configuration parameter if you need to store non-text files in TLIB libraries; see p. 294.

ES command: extract Specified (old) version

To retrieve an old version of your source file from a library file, use the ES (extract specified version) command. This is an example of a “composite” TLIB command; that is, a basic command (“E”) with one or more suffix characters appended (“S”, for “specify version”).

After typing the ES command, you will be prompted for the name of the file you wish to retrieve, just as with the plain E command. If you want to extract `XXX.PAS` from `XXX.P$S`, you would enter `XXX.PAS`. (However, if you wished to avoid a name conflict with the current version of `XXX.PAS`, you could specify another extension, like `XXX.P2S`, so long as the extension you specify still “maps to” the right library file name, `XXX.P$S`).

If `XXX.PAS` already exists, or if `XXX.P$S` does not exist, TLIB will display an error message. Otherwise, TLIB will list the dates and comment lines for all the versions of `XXX.PAS` which are in the library, so that you can select the one you want. Each version's date and comment line will be displayed preceded by a version number (the first is usually version 1), and you will be asked to choose a version:

```
 What version number do you wish to extract ([Enter] for none
)?
```

Type a version number and press `ENTER`. TLIB will re-create the specified version of `XXX.PAS` from the library. As a convenience, you can use “`*`” (asterisk) to refer to the most recent “trunk” (normal development path) version, or “`*-1`” to refer to the next-most-recent.

If you change your mind and decide not to extract a source file, just press `ENTER` alone, without a version number, and TLIB will return to the main menu without creating the source file.

(*Note:* Trunk version numbers are the usual, integer (or *major:minor* pair) version numbers. However, TLIB also allows more complex version numbers. For details, see p. 73, and the discussion of branching on p. 68)

Examples:

```
TLIB ES XXX.PAS 3
```
(extract version 3 *of* `xxx.pas` *to current directory)*

```
TLIB ES XXX.PAS *
```
(extract latest trunk `xxx.pas` *into current directory)*

```
TLIB ES SUB\XXX.PAS *
```
(extract latest trunk `xxx.pas` *into directory* `.\sub`*)*

If you have a “version label” file (a.k.a., “snapshot”) then you can specify the version number as “`@`*filename*”, and TLIB will read *filename* to determine the version number. (If this makes no sense, don't worry about it now; version-label/snapshot files will be explained later.)

Example:

```
TLIB ES XXX.PAS @beta.lis
```
(extract recorded version of `xxx.pas`*)*

You can also retrieve by date & time, instead of version number or version label file. Simply specify the date and/or time (in TLIB's usual date/time syntax) instead of specifying the version number, and TLIB will retrieve the version which was current at the specified time. For details, see p. 241. (Note: TLIB assumes - but does not check! - that the versions are stored in the library in chronological order).

If you already know what version you want, and you don't want to see the list of versions, you can specify the version on the same line when you enter the file name (just separate them with a blank, as you would on the DOS command line).

Command Line Parameters and Batch Files

TLIB is easiest to run interactively, using the GUI interface. However, you can also run command-line versions of TLIB with DOS command line parameters. Almost anything which you can do interactively you can also do using command line parameters. This allows you to build DOS `.bat` files (or OS/2 `.cmd` files) containing TLIB commands.

The rule is that the commands and additional arguments must be specified on the DOS command line in the order that you would have entered them, had you run the command-line version of TLIB interactively. If you specify only some of the required commands and arguments, then TLIB will "lead you through" the rest, with prompts. For example, if you entered:

```
  TLIB N MYFILE.PAS This is my program
```

then a new library file would be created, probably called `MYFILE.P$S`, and the first version would be headed by the comment, "This is my program".

However, if you had entered:

```
  TLIB N MYFILE.PAS
```

then `MYFILE.P$S` would be created as before, but you would be prompted to enter the comment line from the keyboard. If you want to do an N or U command without a meaningful comment (not recommended), you can specify a one-character comment line, like this:

```
  TLIB N MYFILE x
```

To retrieve an old version without being prompted for the version number, simply specify it on the command line. For example:

```
  TLIB ES CALC.C 23
```

Note that you can put as many commands on the command line as will fit, except that no additional commands can appear after a command which creates or modifies the library (N or U), since the additional commands would be misinterpreted as the new revision's comment line.

UF command: Fast Update

The UF command (the U command with the “F” fast/freshen suffix) is similar to the U command except that it will not update a library file unless the library file is older than the source file. This amounts to a limited built-in MAKE facility. It is especially useful when you are updating a large number of source files through the use of wild-cards or file lists.

Note: the UF command is intended for use when check-in/out locking is disabled; that is, for a single programmer working alone. For a similar facility that is more appropriate for group projects, use the UO (update owned files) command.

For example, if you had modified several of the “`.PAS`” (Pascal language) source files in the current directory, you could quickly and easily add all the changes to the corresponding TLIB libraries like this:

```
  TLIB UF *.PAS
```

Note that the UF command, like MAKE, depends upon accurate date and time stamps. If your computer does not have a reliable real time clock, you must either set the date and time religiously every time you start your computer, or else not use the UF command.

L command: List versions

If you wish to see a list of the source file versions contained in a library file, you can use the L (list) command. The list is displayed in the same format as for the ES (extract specified version) command, described on page 37.

The list output can be thought of as an audit trail for the development process, since it documents every change made to the source code, along with when the change was made, and (optionally) who made it. (*Note:* TLIB's appending journal provides another kind of audit trail facility; see p. 190.)

After you enter the L command, you will be prompted for the name of the source file for which you wish to list the versions:

```
  List versions of what source file?
```

If the library file does not exist, TLIB will display an error message and return to the main menu. Otherwise, it will display the file name, date, and comments for each version in the library. If there are too many versions to fit on the screen at the same time, TLIB will pause after each screen full to give you time to read it (but only in interactive mode, not if you specified "`L` *filename*" on the DOS command line).

To print the list of versions (revision history), or save it in a file, you can use command line parameters and DOS file redirection, like this:

```
 TLIB L XXX >PRN
 TLIB L XXX >XXX.LST
```

You can use wild-cards and file lists to easily print a module-by-module report of the revision history for several modules or even the entire project; see p. 50.

CP command: override library Path

You can, if you wish, keep your library files in the same disk directory as your source files. More often, however, library files are stored in a different hard disk directory, or perhaps on a network file server or on a diskette in a different drive.

TLIB provides the `PATH` configuration parameter for this situation: it allows you to specify the path (drive and/or subdirectory, or a “UNC” path) in which your library files reside. See p. 257 for details.

Sometimes, however, you may wish to override the `PATH` configuration parameter and force TLIB to look in a different place for the library file(s). For this, TLIB provides the CP (configure library path) command.

For example, if your source file is in the current directory on `D:`, and your library files are on a diskette in drive `B:`, you could give the CP command and enter “`B:\`” when TLIB asks you for the library file's path.

Like all TLIB commands, the CP command can also be specified on the DOS command line. For example, to update the library file on `B:` with the latest verson of `D:MYFILE.TXT`, you could type:

```
  TLIB CP B:\ U D:MYFILE.TXT
```

This makes it simple to prepare a tiny batch file to do the CP command each time you run TLIB. For instance, you might use a batch file called `BLIB` when your TLIB libraries are on the `B:` drive:

```
 REM blib.bat: batch file to run TLIB
 REM with libraries on the B: drive
 TLIB CP B:\ %1 %2 %3 %4 %5 %6 %7 %8 %9
```

Then to update the library file on `B:` with the latest verson of `D:MYFILE.TXT`, you could type:

```
 BLIB U D:MYFILE.TXT
```

The CP command can also be used to specify the complete library file name, rather than allowing the library file name to be determined by the name of the source file. The file name must contain (or be followed by) a

“.” (period), since this is how TLIB distinguishes the file name from the name of a directory.

This form of the CP command is seldom used, since it requires that you use a separate CP command for each library file. You will generally find it easier to let TLIB determine the library file name from the source file name, instead of using this form of the CP command. Nevertheless, here are a couple of examples:

```
  TLIB CP MYLIB.XYZ
```

(The library file is `MYLIB.XYZ`, in the same directory as the source file.)

```
  TLIB CP C:\LIBS\MYLIB.
```

(The library file is `MYLIB` (with no extension), in directory `C:\LIBS`.)

The CP command accepts exactly the same path specifications as the `PATH` configuration parameter, including several rather obscure variants. For instance, you can specify a list of directories for TLIB to search, or you can force TLIB to store “lock files” (when check-in/out locking is enabled) in a different place from its library files. For an in-depth treatment, refer to the `PATH` configuration parameter, p. 257.

Text File Formats

Tabs

If you use the default configuration parameters, TLIB cannot recognize the equivalence of tabs and blanks. That is, tab and blank are considered to be two different distinct characters.

However, if you configure `DETABE M` and `ENTABU Y`, then tab characters are equivalent to the appropriate number of blanks (with tab-stops assumed at columns 9, 17, 25, 33, etc.). This allows the library file to be kept as small as possible by converting multiple blanks to tabs. When a source file is extracted from its library file, the tabs are converted back to blanks. For details, see below and p. 255.

If you wish to convert sequences of multiple blanks into tabs, or expand tabs into multiple blanks, you can use the TABS utility. TABS is included with TLIB. Like TLIB, it was carefully written to be very fast. However, at this writing it only supports “8.3” (short) file names.

To shrink a file by replacing multiple blanks with tabs, do:

```
  TABS IN oldfile newfile
```

To reverse the process, that is, replace tabs with blanks, do:

```
  TABS OUT oldfile newfile
```

You can also convert 4-space tabs to blanks (but not vice-versa):

```
  TABS OUT4 oldfile newfile
```

Do not try to run TABS with the same file as both input and output.

Note that you can use wild-cards (asterisks) in both input and output filenames. For example:

```
 TABS IN *.c *.new
```

Configuring blank/tab treatment

Some TLIB users need to configure TLIB for automatic tab/blank conversions, but most do not.

TLIB's tab/blank conversion algorithms are the same as those used by the DOS `TYPE` and `PRINT` commands (i.e., tab stops at columns 9, 17, 25, etc.). So, one way to decide whether you need tab/blank conversions is to use the DOS `TYPE` or `PRINT` command to display a few of your files.

In general, if your files look right when displayed with the `TYPE` and `PRINT` commands, then you can probably enable tab/blank conversions; but if:

the indentation is wrong,

or if the lines are broken in the wrong places,

or if columns don't line up correctly,

then you probably should *not* enable TLIB's tab/blank conversions.

More specifically:

1) If you use an editor (like IBM's old *Personal Editor II*) which silently converts between blanks and tabs, you *should* enable TLIB's tab/blank conversions. Otherwise, if your editor changes blanks to tabs or vice-versa, then when you add the new version to your TLIB library file, TLIB will think that almost every line has changed (so the “delta” added to your library file will be huge).

2) If you use an editor which lets you choose the positions on the line at which your tab stops will fall, and does not convert the tabs to blanks when you save the file, and you use non-standard tab stops (e.g., 3 or 4 spaces per tab), then you must *not* enable TLIB's tab/blank conversions. The popular *MultiEdit* editor from American Cybernetics can be configured like this; so can most word processors. However, *MultiEdit* can also be configured to convert tabs into spaces; if you use *MultiEdit* like this, then you *can* enable TLIB's tab/blank conversions.

3) If you use TLIB to manage files containing more than 254 characters per line, then you must *not* enable TLIB's tab/blank conversions. TLIB cannot properly convert between tabs and blanks in files which contain lines of more than 254 characters.

Lines that long most commonly occur in word processing documents, such as “Rich Text” files, and files produced by *XyWrite III+*. XyWrite (and some other word processors) use Carriage Return / Line Feed to delimit paragraphs instead of lines; this results in “lines” (really paragraphs) which often exceed 254 characters.

Note: Many word processors, such as *WordPerfect* and *Microsoft Word*, use special *binary* formats to store documents. To handle these files with TLIB, you must configure “`FileType Runlen`” or “`FileType Binary`” (or “`FileType Auto`”), and TLIB will ignore the `ENTABU` and `DETABE` settings.

Note: The *StarOffice* and *OpenOffice* word processors are special cases. Their document files (`.sxw`, etc.) are actually compressed (“zipped”) archives. To store one of these files under TLIB, we recommend that you first rebuild the `.sxw` archive with decompressed (“stored”) xml components. Please contact support@burtonsys.com if you need a batch script to simplify this.

4) If you are using TLIB to manage files which *must* contain tab characters, then do *not* enable TLIB's tab/blank conversions. For example, a few MAKE utilities require that certain lines in your “makefile” begin with a tab character. (Note: Opus Make does not have this requirement.)

To enable tab/blank conversions, add the following to the end of your TLIB configuration file (probably `TLIB.CFG`), or change the `ENTABU` parameter configured by TLIBCONF or the TLIB Configuration Wizard:

```
  ENTABU Y
  DETABE M
```

To configure varying tab treatments for different source files, or to indicate that some source files need to be stored in binary format, you can use TLIB's `IF` and `ENDIF` parameters, as in the following example:

```
 DETABE M
  ENTABU N
  IF *.ASM
    REM - Do tab/blank conversions only on assembler files
    ENTABU Y
  ENDIF
  IF *.WK*,*.LIB
    REM - Spreadsheets and .lib files are not ASCII text
    FILETYPE BINARY
  ENDIF
```

End-of-file markers

DOS had two ways of marking the end of a text file. The first method simply set the file's length to be the total number of characters in the file. The second way was similar, except that there was one extra character appended to the end: a Ctrl-Z (ASCII code 26).

The us of a CtrlZ as an end-of-file indicator is largely obsolete, but TLIB will accept either format as input. However, all text files created by TLIB use the first format: there will not be a Ctrl-Z at the end (unless you configure `ADDCTRLZ Y`, see p. 276).

Managing Multiple Source Files

Most programs are built from a large number of source code files. This leads to several common questions.

The first question for developers using many source files is, “how can I conveniently do an operation on a whole set of source files?” GUI versions of TLIB let you do this by selecting the files in a “pick list” window. Plus, both GUI and command-line versions of TLIB support wild-cards and file lists.

Wild-cards and file lists can be used in any command where you specify the name of a source file, e.g., the U (update), E (extract), N (new library), and L (list) commands.

An example of the use of wild-cards in a U (update) command is:

```
  TLIB U *AWK*.C
```

TLIB would do a U (update) command on every `.C` file in the current directory whose names contain “`AWK`”, for example, `MYAWKS.C`, `AWKFILE.C`, etc..

File lists are used similarly:

```
  TLIB U @myfiles.lis
```

TLIB would do a U (update) command for every file with a name listed (one file name per line) in the text file, `myfiles.lis`.

Another question to consider is, “how can I be sure that I've updated all the relevant library files” after revising a group of source files? If you work alone (and so you do not need to use check-in/out locking), you can just use wild-cards with the U (update) command, like this:

```
  TLIB U *.c,*.h
```

LIB will say “`no changes`” for each unchanged source file, and prompt you for a comment line for each modified source file. If you want to skip a file, without doing the Update, just press `ENTER` without typing a comment line.

You can speed up this process by telling TLIB to skip source files that are older than the corresponding libraries, by using the F (fast/freshen) suffix, like this:

```
  TLIB UF *.C,*.H
```

This is a very fast and simple way to add all of your latest changes to the library files; see also p. 40.

In a networked development environment, with check-in/out locking enabled, `LOCKING Y` (to coordinate multiple programmers), you can do much the same thing, but you would use the `UO` (**U**pdate **O**wned files) command instead of the `UF` command. (The “O” in “UO” is the letter O, for “owned,” not the numeral 0.) The O (owned) suffix limits the scope of the update command to just those files that you have checked-out/locked. O (owned) is a wild-card search mode suffix; for more about wild-cards and search modes see pp. 50 and 59.

A third problem for developers is, “How can I easily retrieve a *particular version* of each of a set of related source files?” This is known as the “version labeling” problem. TLIB solves it with the S (“snapshot”) command, for labeling the set of modules and versions in a particular release; see p. 92. (Note: the S command replaces the old **TLIBSNAP** program which was included with TLIB 4.12.)

Additionally, if you need to manage multiple actively-changing versions of a single program, you can use TLIB's *Named Project Levels*. For example, you may be supporting one or more old releases of a software product as you develop the next release. Or, you may be managing “semi-custom” software, in which you have one standard version and many customized variants of it, all of which must be continually maintained. Version tracking and named project levels are described starting on p. .

Wild-cards and File Lists

Wild-cards can be used in any command where you specify the name of a source file, e.g., the U (update), E (extract), N (new library), and L (list) commands.

An example of wild-cards in an E (extract) command is:

```
  TLIB E *AWK*.C
```

TLIB would extract the current version from each of the libraries.

Note that you can use as many asterisks and/or question marks as you wish. Additionally, TLIB supports six different wild-card "search modes" to select files in various ways; see p. 59.

Or, you can use a file list. File lists are very simple. They are usually just ASCII text files containing file names, one per line. File lists are used similarly to wild-cards. For example, if you had a file called `MYSTUFF.LIS` containing the lines:

```
  MYFILE.C
  MYJUNK.C
```

then this command would update the library files for both files:

```
  TLIB U @MYSTUFF.LIS
```

You could also do:

```
  TLIB E @MYSTUFF.LIS
```

to extract `MYFILE.C` and `MYJUNK.C`.

The GUI versions of TLIB can create file lists for you if you use the "Plain File List" option of the Snapshot command. Additionally, TLIB's **LISTBLD** utility can build file lists *automatically*, by scanning your source code for “include” references to other modules, and by combining and manipulating other file lists; see p. 203.

LISTBLD can build file lists for many languages, but not for xBase. However, *dBFind*, a dBase syntax checker from Sigma Six (formerly The Software Development Factory) has this capability. dBFind can automatically generate TLIB-compatible file lists, by analyzing dBase or Clipper source code. Sigma Six is at P.O. Box 1106-B, Hunt Valley, MD 21030, Tel: 410-666-8129, http:/www.sigmasix.com/. (Tell Steve Johnson we said hello.)

For compatibility with *CrossRefC* (a C cross-referencer tool) from Sigma Six, we allow an optional trailing “+” after each file name, at the end of each line in a file list; however, it is not required.

You can also put version number specifications in your file lists, if you wish. Such file lists can be used as “version labels” to record which versions of each of a collection of files go together. However, version label files are rarely created manually. Instead, you'll probably use TLIB's S (snapshot) command to create version labels; see p. 92.

You can put comments in your file lists. Simply precede each comment line with an exclamation point and blank or two exclamation points (“`! `” or “`!!`”) in the leftmost two columns.

If you have files with names that begin with “!!”, you can list them in your file lists with an explicit directory specifiction, like this:

```
 YOURJUNK.C
 .\!!MYJUNK.C
```

Restrictions

There are several restrictions on the use of wild-cards and file-lists with TLIB. They are:

o You cannot use wild-cards to specify multiple file lists. That is, you cannot use specifications like `@*.lis` o File lists can be nested, but only up to 3 levels deep. That is, file lists can contain specifications like `@ name.ext`.

o When used in most contexts, file lists can contain wild-card specifications. That is, file lists can contain specifications like `*.c` and `*.asm`. (Exception: you can't do this with branch version specifications. If this makes no sense, don't worry. Branch versions will be covered later.)

o Some versions of TLIB have limited support for maintaining TLIB library files within PKZIP-archives. However, TLIB does not allow the use of wild-cards with archived (PKZIPed) library files. You can use file lists with archived library files, if the file list does not contain wild-card specifications.

DOS errorlevels

When processing a single file, command line versions of TLIB return a DOS errorlevel of zero for success or non-zero for failure, except that there are special rules for the T (test lock status) command.

When processing multiple files (via wild-cards and/or file lists), command line versions of TLIB return an errorlevel which is the maximum of the errorlevels which would have been reported for any of the individual files, had such a file been processed by itself (without wild-cards). This has the effect of ensuring that if an error occurred for any of the processed files (and the error was severe enough to have caused a non-zero error level), then the errorlevel will be non-zero.

TLIB for Windows shows the errorlevel result for each file in the `status` column in the main window.

Comma-Delimited Lists of Files

TLIB also supports “in-line” comma-delimited file lists. That is, you can do commands like:

```
  TLIB uf *.c,*.h Fix PTR no. 8/159
```

Important: do *not* include any blanks in the list of file specifications. The following will *not* work:

```
  TLIB uf *.c, *.h This won't work!
```

the comment is “`*.h This won't work!`“ The problem is that “`*.h`” is part of the comment.

You can even include `@file.lis` references within an in-line file list, and vice-versa:

```
  TLIB uf @file1.lis,@file2.lis This is a comment
```

Even if you specify a file multiple times, it will only be processed once. When processing multiple files, TLIB first expands the wild-card specifications and file lists into a single internal list of files, and eliminates duplicates, to avoid processing any file twice. This mechanism works when using both wild-cards and file lists (both kinds: files containing file names, and in-line comma-delimited lists).

This is convenient when using file lists that contain wild-cards, which might otherwise specify the same files multiple times in different ways. For example, the file `myprog.c` is specified twice in the following command, but will only be processed once:

```
  TLIB u *.c,*.h,myprog.* Fix PTR no. 8/159
```

This also avoids idiosyncratic behavior when extracting files onto a Novell Netware drive using wild-cards with a search mode that matched the extracted files: earlier versions of TLIB would extract each file twice because Novell's “find-first/find-next” search implementation (unlike

DOS's) does not always consider an extracted file to be the same as the file it replaced, and so it finds the file name twice. (We don't know why it was *only* twice, however.)

Integration With Other Products

TLIB has a variety of integration features, to make it easier to use with other products.

“Native” Project File Support

TLIB has built-in support for the compiler-native project files of many programming languages and other development tools.

To access this feature from within TLIB for Windows, click the “Mode” toggle button. You can then directly “open” a native-format “project file” used by any of ten popular development tools (plus HTML and .shtml web site files), instead of using wild-cards and file-lists to specify source files.

To access this feature from the command-line versions of TLIB, reference the compiler's project file as if it were a TLIB file list, except that you should specify “`@@`” (two @s) before the name of the project file. For example, you could store all the source files in a VB 5.0 program called `myproj` with this command:

```
 tlib nf @@myproj.vbp
```

Currently we support Microsoft Visual Basic 3.0 `*.mak` and VB 4.0/5.0/6.0 `*.vbp` projects, Microsoft Developer Studio / VC++ 5.0-6.0 projects and workspaces (`*.DSP` & `*.DSW`), Borland Delphi projects (`*.dpr`), Borland C++ Builder projects (`*.bpr`), Watcom C/C++ 10.5 project (`*.wpj`) and target files (`*.tgt`), Symantec Visual Cafe projects (`*.prj`), and Help Magician Pro (`*.hmp`). Also, we support the project files of three popular programmers' editors: MultiEdit (`*.PRJ`), Visual SlickEdit (`*.VPJ`), and CodeWright (`*.PJT`). Also, we support using a “main” web site file (typically `index.html`, `index.htm`, or `index.shtml`) as the project file for a web site (TLIB parses the file and follows the links to find the rest of the web site's files).

You can also specify a bit-sensitive options value, within square brackets at the end of the project file name. The bit values supported vary by project file type, but the two most common ones are: Bit 0 set to ‘1’ means that certain kinds of binary files found in the project are subject to version control; bit 1 set to ‘1’ means that TLIB should scan for `#include` directives so that the list of files deduced includes the header files.

For example, to store the initial version of a Watcom C/C++ project into TLIB, scanning for `#include`, you could type:

```
 tlib nf @@myproj.wpj[2]
```

This feature is not perfect. Some tools have idiosyncrasies that are difficult to manage, such as “adjustable” include file search algorithms, and Watcom C/C++ include file scanning is necessarily a bit sluggish.

For the most part, TLIB's include file searching algorithm for Watcom C/C++ projects mimics that of the Watcom compiler, except that:

o TLIB ignores system header files (files referenced with an `#include` directive that uses `<angle brackets>` to quote the header file name. (This is because it is rarely useful to store the system header files under TLIB.)
o TLIB honors the `-i` include directory list(s) that are stored in each target (`.tgt`) file, except that TLIB ignores any include directories that reference Make macros, such as "`$(%watcom)`" or "`$[:`".
o TLIB ignores the `%{os}_INCLUDE%` and `%INCLUDE%` environment variables.
o Watcom's description of their include search algorithm specifies that the first place looked is the "current" directory, followed by the directory containing the "parent" file (that is, the including file, the file with the `#include` directive), followed by the parent file's parent file's directory, etc.. TLIB follows this algorithm exactly, but note that the "current" directory in this context is assumed by TLIB to be the directory that contains the target (`.tgt`) file.

We plan to add support for other languages, as well. If your favorite language uses project files that are not supported by TLIB, please give us a call (better: send us email to `support@burtonsys.com`), and perhaps we can add support for it.

Using native project files is very convenient in the TLIB for Windows GUI: just click the “Mode” button, and use the “open” dialogue, and all the complexity of wild-cards search modes, etc., disappears. In command-line versions of TLIB, the lack of a pick-list for selecting among the files in a project makes compiler-native project file support less powerful.

APIs for Integration with Windows Programs

TLIB for Windows is constructed in two parts: with a GUI “front end” user interface, and a DLL “back end” version control engine. The API (Application Program Interface) with which the GUI front end communicates to the DLL is also available for use by other programs. There are both 16-bit and 32-bit versions; the interface specification is in `TLIBDLL.H`.

The 32-bit GUI version of TLIB for Windows also supports a “mailslot-based” interface, which allows other Windows applications to easily integrate with TLIB for Windows via simple mailslot reads and writes. This is usually a simpler alternative than calling the back-end DLL API

Additionally, we are in the process of implementing a Microsoft Source Code Control interface DLL; at this writing, it is partially working, but unreleased. Please contact us if you need it.

Add-In for Visual Basic

TLIB for Windows (included in the TLIB Combo Edition) has add-ins for Visual Basic 4.0 (32-bit version), 5.0, and (beta) 6.0, to hook into the VB IDEs. The Add-In for Visual Basic 5.0, works with the Professional, Enterprise, and Learning Editions of VB 5.0, but not with the control-creation edition. We also have a beta Add-In for Microsoft Developer Studio 6.0 (VC++, etc.)

Plus, TLIB can directly “open” VB 3.0, 4.0, 5.0, and 6.0 project files, as described below. (VB.NET project groups will be supported soon, call or email us if you need this feature.)

Borland Delphi and C++ Builder

TLIB can directly “open” Borland Delphi and C++ Builder project files, as described above.

Watcom C/C++

All editions of TLIB include batch scripts `GEN_CI.*` and `GEN_CO.*`, for hooking TLIB into the Watcom IDE. See the file `WATCOM.TXT` for details.

Plus, TLIB can directly “open” Watcom C/C++ 10.5 project and target files, as described below.

Microfocus COBOL Workbench

See `MFCOBOL.ZIP`.

Programmers' Editors

The “big three” programmers' editors, MultiEdit, CodeWright and SlickEdit, are all integrated with TLIB. So is Brief. Others (such as ED and The Semware Editor) may also have integration with TLIB by the time you read this. See the editor's documentation, or call Burton tech support if you need help.

Six Different Wild-card Search Algorithms

TLIB's wild-card searching has been significantly enhanced. With TLIB 5.50, you *always* specify the file names (or wild-card specifications) for the *source* files (instead of for the library files). However, you can instruct TLIB to use any of six different wild-card search algorithms, by appending any of six suffix letters to those TLIB commands which accept source file names: N, U, E, L, T, S, A, M.

The first six suffixes below are called “search mode suffixes.” In addition, TLIB supports two special suffixes, N and I:

Suffix	Meaning
`W` “Work files”	searches the existing normal source files, like the DOS “`dir`” command would do with the same wild-card specification.
`L` “Library”	searches for the corresponding library files.
`C` “Checked-out”	searches for the corresponding lock files (which normally exist only for files which someone has checked-out/locked).
`O` “Owned files”	like “`C`”, but only those files which you have checked-out/locked.
`T` “This project lev”	searches for names defined in the current project level (if any). The current project level is selected via the `PROJLEV` configuration parameter.
`A` “All levels”	searches for names defined in the current project level and/or its predecessor level(s), as determined by the `LEVEL` configuration parameter for the current project level.
`N` “No search mode”	a special suffix used only with explicit (non-wild-card) specifications, to disable directory look-ups even if `FIND1FILE Y` is configured.
`I` “Include subdirs”	add this suffix to make TLIB scan subdirectories (if `TREEDIRS Y` is configured).

Note #1: as is probably obvious, the C and O suffixes only work if you have locking enabled (`LOCKING Y`, `LOCKING B`, etc.), and the T and A suffixes only work if you have tracking enabled with named project levels (`TRACK Y`, `PROJLEV` *name*, `LEVEL n=`*name*...).

Note #2: the wild-card search mode is of consequence only when you use wild-cards to specify your source files, or when `FIND1FILE Y` is configured (since `FIND1FILE Y` causes all file specifications to be interpreted as if they were wild-cards).

Most of the wild-card search modes can be combined, as well. When you specify more than one search mode, TLIB expands the list of files for each search mode, and combines the lists, removing duplicates. The I (include subdirs) suffix can be specified in combination with any of the six search modes. The N (no search) suffix can only be specified by itself.

In TLIB for Windows, the search mode options are provided by check boxes on the main screen.

The search mode suffixes can be specified in combination with the other command suffixes:

Suffix	Meaning	Applicable commands
`B`	Browse mode	`E`
`D`	Discard lock	`U`
`F`	Fast/freshen	`E,N,U,A`
`K`	Keep checked-out	`N,U`
`M`	Minor version number increment	`U`
`R`	Reserve lock	`E`
`S`	Specify version number	`E,N,U,S,M`

The order in which suffix characters are specified is inconsequential. For example, EBT and ETB are exactly the same command (browse-mode extract, with wild-cards matching the file names listed in the current project level).

The default search mode depends upon the command:

Command	Meaning	Default search mode
`N`	New library create	`W` (work files)
`U`	Update library	`W` (work files)
`L`	List versions	`L` (library files)
`E`	Extract source	`L` (library files)
`S`	Snapshot	`L` (library files)
`M`	Migrate	`L` (library files)
`T`	Test lock status	`C` (lock files)
`A`	Add/Alter project level	`A` (all project levels)
`AP,AD`	Promote or Delete	`T` (this project level)

Important note #1: The L, C and O search modes are dependent upon the `EXTENSION` configuration parameter (p. 254).

Note #2: The `TREEDIRS` parameter determines whether or not you can use the I (include subdirs) option. If you've configured `TREEDIRS Y` then you can make TLIB search subdirectories by appending the I suffix (or checking the Include Subdirs box in TLIB for Windows).

For example:

`TLIB ETI *.c`	extracts all `.c` files listed in current projlev
`TLIB ET *.c`	extracts only the `.c` files for current directory

Note #3: The search mode suffixes may not appear in the prompt and help screens which are generated by TLIBCONF or the TLIB Configuration Wizard.

Note #4: Disabling search modes in command-line versions of TLIB: TLIBCONF (for DOS-only versions of TLIB) and the TLIB Configuration Wizard (for the combo version) normally configure TLIB with all commands and all search modes enabled. Because of the very large number of different possible TLIB commands which can be built by combining the various search mode suffixes and regular suffixes with the basic commands, we do not to require that you list all the combinations of commands and search mode suffixes in TLIB's `COMMANDS` configuration parameter. Instead, if you list a command without any search-mode suffix in the `COMMANDS` configuration parameter, then all search mode suffixes will be allowed for that command. If you wish to allow only particular search mode suffixes for a command, you may do that, too, but for that command you must list all the combinations that are to be allowed, including the search modes (specify the suffixes in alphabetical order).

TLIB 5.50's handling of wild-cards is enhanced in the following ways from TLIB 5.00:

o The I suffix controls whether or not subdirectories are searched (when `TREEDIRS Y` is configured).

o With earlier versions of TLIB, only the T and A search modes were capable of processing subdirectories, and whether they did so or not was determined by whether or not you specified a specific directory. Neither of these statements is now true. All TLIB commands operate on a just a single directory, regardless of the search mode(s), unless you add the the I (Include subdirectories) option letter to the TLIB command, to make it process subdirectories (if you've configured `TREEDIRS Y`). This behavior is now consistent regardless of what search mode(s) you use.

o TLIB now fully determines the list of files that a command will operate upon before it processes the first file. (Earlier versions of TLIB processed the files as the names were determined.) This means that if you have a lot of files there may sometimes be a noticable pause before the first file is processed, especially if you use the `I` suffix to process subdirectories, and especially if you use several wild-card search modes in combination.

o You can now combine two or more search mode suffixes to make TLIB find all files that would be found by any of the search modes. For example, if you have work files for abc.c and def.c, and there are TLIB libraries for `DEF.C` and `GHI.C`, you can use `LW` as the wild-card search mode to select all three files (the "W" makes TLIB find files `abc.c` and `def.c`, the "L" finds makes TLIB find `def.c` and `ghi.c`, so "LW" makes if find all three). When you specify more than one search mode, TLIB simply handles each search mode in turn and merges the resulting lists of files.

(We mainly added the multiple-search-mode support so that TLIB for Windows could have a default search mode that would be reasonable regardless of which command was to be used. This was needed because the GUI front end of TLIB for Windows uses a paradigm in which the user first selects the files to be processed, then selects the operation to be done.)

o You can now ask TLIB to tell you what files it would operate upon, without actually doing the operation. Simply preceed the command in question with “`G1`”. Thus, `TLIB G1EBFL *.C` will list the files that would be extracted by the `TLIB EBFL *.C` command (Extract Browse-mode Freshen with the "L" search mode).

F - Filter File Names

In TLIB's GUI Windows user interface, you can select which source files TLIB will operate upon before issuing the TLIB command. However, in command-line versions of TLIB, you don't have that flexibility. Instead, you must specify the source files to operate upon by using wild-cards and/or file-lists.

The F (Filter file names) command help improve the flexibility of command-line versions of TLIB, for specifying source files to be processed by TLIB. The F command lets you specify a wild-card specification (or several of them, separated by commas) which TLIB will use to “filter” the file names found when processing subsequent commands.

The F (filter) command is mainly useful in command-line versions of TLIB. In the TLIB for Windows GUI, the main file pick list offers a similar (but more flexible) capability, so F (filter) is not needed (though it is accessible via “Run Manual Command”).

The F (filter) command is mainly intended for selecting particular files from file lists, though you can also use it as an “AND-clause” when specifying files with wild-cards.

For example, this command would list the versions of all the `.C` and `.H` files named in the file list `ALLFILES.LIS`:

```
 TLIB F *.c,*.h L @allfiles.lis
```

The filter's wild-card specifications are of the same sort as those which you can use in `IF` clauses (p. 321) in your TLIB configuration file: they can contain one or more wild-card file specifications separated by commas. The wild-card specifications can have multiple, leading and/or embedded asterisks and/or question marks in them, if you wish. However, you cannot include directory specifications or the `@filelist.ext` format.

Examples:

`TLIB F myfile.* E @all.lis`	this is okay
`TLIB F *my*.* E @all.lis`	this is okay
`TLIB F make*.*,*.c E @all.lis`	this is okay
`TLIB F *.c,@xyz.lis E @all.lis`	illegal (“`@xyz.lis`” prohibited)
`TLIB F mydir\*.c E @all.lis`	illegal (“`mydir\`” prohibited)

Once you have specified a filter, it remains in effect until you either quit TLIB, or until you explicitly change or disable the filter.

You can specify a period as the filter to disable file name filtering. For example, in the following TLIB command line, the `*.c` filter would apply to file names found in `ALL.LIS`, but no filter would be used when taking file names from `BATS.LIS`:

```
TLIB F *.c E @all.lis F . E @bats.lis
```

Note #1: If you upgraded to TLIB 5 from TLIB 4, then your existing `TLIB.CFG` file may define “F” as a shorthand for “UF” (fast-update/freshen), via the “`COMMANDS` ...`F=UF,`...” configuration parameter. (TLIBCONF can optionally configure TLIB this way if you say that you want TLIB 5 to use TLIB 4's single-character commands.) If this is the case, then you can either remove “`F=`” from the `COMMANDS` parameter, or else you can specify `F0` (“F zero”) instead of `F` when you want to use a filter.

Note #2: Because TLIB for Windows lets you expand wild-cards and then select files from the expanded list, the F (filter) command is rarely needed, so we did not include it on the main screen. To use the Filter command in the TLIB for Windows GUI, select “Run” and enter a manual command of “`F` *wildcard-specs*”. The filter will persist until you reset the current directory or exit TLIB for Windows.

Tree-structured directories

TLIB supports keeping your TLIB library files (and/or lock files) in a "tree" directory structure which mimics that of your source files.

To use this feature, you must do three things:

i) Configure `TREEDIRS Y`.

ii) Change your `PATH` configuration parameter (or CP command), adding "`*\`" at the end. TLIB will substitute the "relative path" of the source file (relative to `WORKDIR`, your main work directory) for the "`*\`" in the path.

iii) Enable "version tracking" for your source files (version tracking is explained later).

For example, consider the following TLIB configuration file excerpt:

```
TREEDIRS Y
LIBEXT ??X
LOKEXT ??Z
PATH F:\TLIBS\*\
IF *.c,*.h
  TRACK Y
ENDIF
```

Then if your main work directory is `C:\WORK\`, and you have a source files named `C:\WORK\IOSTUFF\MYFILE.C` and `C:\WORK\MAIN.C`, you could create TLIB libraries for them like this:

```
cd \work
tlib n iostuff\myfile.c This is a comment
tlib n main.c This is a comment
```

TLIB will create these library files:

```
F:\TLIBS\IOSTUFF\MYFILE.C_X
F:\TLIBS\WORK.C_X
```

Note that the `F:\TLIBS\IOSTUFF` and `F:\TLIBS` directories should already exist; TLIB will not create them for you (unless you configure `MAKEDIRS Y`, see p. 315).

Branching: Version Trees

With branching, your version numbers can contain up to 9 decimal points, so that, for instance, if your library contains versions 1 through 15, you can later go back and add version 9.1. Version 9.1 is called a *branch* version; versions 1 through 15 are called *trunk* versions.

You might add version 9.1 if, for instance, version 9 was part of an earlier release of your program and you needed to make a minor bug fix to it. Version 9.1 is not considered the "newest" version in the library, even though it may have been added after version 15.

Note well: Version 9.1 is *not* in any sense "between" versions 9 and 10! Do not think of the dot in "9.1" as a decimal point. Rather, the dot is just a separator character. Version 9.1 is "beside" or "in parallel with" version 10. Version 9.1 may be either newer or older than version 10, but it is *not* an ancestor of version 10. Instead, both version 9.1 and version 10 have version 9 as their immediate ancestor.

Most TLIB version numbers are simply consecutive integers. These are the "trunk" versions. The trunk version after version "5" (if it exists) is usually called version "6" (or, perhaps, the *major:minor* pair "5:1"). Version numbers containing decimal points are called "branch" versions. Version 5.1 would be the first version in a branch from version 5.

If you never have to create "maintenance releases" (bug fixes to obsolete versions), nor maintain parallel lines of development, then you will probably always/only work on the latest versions of your source files, so you will probably never need to use branching.

The "trunk" vs. "branch" terminology derives from the pictorial hierarchy of version numbers. Consider a library file containing seven versions, numbered one through seven, like this:

```
 oldest                                              newest
   1 ——>   2 ——>   3 ——>   4 ——>   5 ——>   6 ——>   7
```

The arrows represent a development time line. Version 7 was derived from version 6, which was derived from version 5, etc.. The versions are in a straight sequence, like the trunk of a tree, growing from left to right (version 1 could be called the “root”).

Usually, the next version would be called version 8, and it would be created by editing a copy of version 7 to incorporate some new improvement.

Suppose, however, you need to make a minor revision to version 3, rather than version 7. You would retrieve version 3 from the library file with the ES command, make your changes, and add it to the library file. However, it really shouldn't be added as version 8, since it is not derived from version 7. Instead, you would like to add it as version “3.1”, since it (like version 4) is derived from version 3.

If version tracking is enabled (`TRACK Y`), then TLIB will create the branch *automatically* because it knows that you started with version 3 rather than version 7. Otherwise, you can use the **UB (update branch)** or **US (update specified version)** command to do this. US is very similar to U (update library), except that it allows you to specify the version number which you wish to create.

For example, if the program was called `MYPROG.PAS`, then you could let TLIB automatically determine the new version number like this:

```
  TLIB U MYPROG.PAS this is my comment line
```

Or, you could add (branch) version 3.1 like this:

```
 TLIB US MYPROG.PAS 3.1 this is my comment line.
```

Alternately, you can use an asterisk instead of the “1” in “3.1”, and TLIB will figure out the actual version number, like this:

```
 TLIB US MYPROG.PAS 3.* this is my comment line.
```

Both of these forms are equivalent unless version “3.1” already exists, in which case the first form would result in an error message, but the second form would create version “3.2”.

Our library file could now be depicted graphically like this:

```
oldest      (1-7 are trunk versions)         newest
  1 ——>  2 ——>  3 ——>  4 ——>  5 ——>  6 ——>  7
(root)          │
                └——> 3.1 (branch version)
```

If version 3.1 of your program didn't work correctly, you would need to revise it again, and you could create version 3.2, and then versions 3.3, 3.4, 3.5, ..., 3.9, 3.10, 3.11, etc.. You could even go back and create versions 3.1.1, 3.1.2, etc..

You can also create additional branches from version 3, “in parallel with” version 3.1, 3.2, etc.. The second branch from version 3 has version numbers which contain a parenthetical “branch number” of “2” (the number of the branch, as opposed to the number within the branch). The first such version, in the second branch from version 3, would be number 3.(2)1, and the library file could be depicted like this:

```
oldest      (1-7 are trunk versions)         newest
  1 ——>  2 ——>  3 ——>  4 ——>  5 ——>  6 ——>  7
(root)          │
                ├——> 3.1 (branch version in 1st branch)
                │
                └——> 3.(2)1 (branch version in 2nd branch)
```

If you go overboard with branching, the structure of your library file can become very confusing, resembling a sideways tree, like this:

```
oldest                                        newest
  1 ——>  2 ——>  3 ——>  4 ——>  5 ——>  6 ——>  7
(root)          │                    │
                ├——> 3.1 ——> 3.2     └——> 6.1
                │      │
                │      └——> 3.1.1 ——> 3.1.2
                │               │
                └——> 3.(2)1     └——> 3.1.1.1
```

We don't recommend creating library files that look like this!

The `UB` (create a new Branch) command tells TLIB that you want to store the new version as a new “something.1” branch, but let TLIB determine the particular branch number. Like the US command, the UB command can be used to create a new branch regardless of whether or not you started with the latest trunk version. However, when tracking is enabled the UB command will always create a new version that is a successor of the version that you started with (unlike the US command, which will create whatever new version number you tell it to create).

Note: you cannot combine the U (update) command's "B" suffix (create branch) with either the "S" suffix (specify new version) or the "M" suffix (increment minor trunk version).

The TLIB E (extract) command (and the EB command, extract for browse) will generally retrieve the “current” version of a file. In the simplest case, this is the latest “trunk” version. However, if you use TLIB's named project levels, then it may be some other version, since which version is current depends upon which project level is your current project level (the current version numbers for each file are recorded in the current level's tracking file -- but that is explained later).

(There is one exception to this rule. If you use a file list or snapshot file to specify the extracted source files, the version number, if any, which is listed in the file list or snapshot file will override the “current” version number.)

To retrieve something other than the current version, use the ES (extract specified version) command. For example, to retrieve version 3.1 of `MYPROG.PAS`:

```
TLIB ES MYPROG.PAS 3.1
```

You can use an asterisk in place of the last part of the version number to mean “most recent.” In the overly complex tree above:

```
TLIB ES MYPROG.PAS 3.*      (retrieves version 3.2)
TLIB ES MYPROG.PAS 3.1.*    (retrieves version 3.1.2)
```

Note that “2.0” is the same as “2”, so if there is no version 2.1:

```
TLIB ES MYPROG.PAS 2.*      (retrieves version 2)
```

The “S” in the “ES” command is an example of a *command suffix*. Most of TLIB's basic commands support one or more command suffixes, which modify the bahavior of the basic command. (In TLIB for Windows, option check-boxes are used in place of the command suffixes.)

The S suffix, as it happens, modifies the behavior of the E command to make it accept an explicit version specification. Other suffixes modify it in other ways. Suffixes that you can specify with the E command are B (browse), F (fast/freshen), and R (reserve). Plus, there are seven other suffixes that can be used to alter the way TLIB interprets wild-card specifications; these are called “search mode suffixes” (p. 59).

Most of the suffixes can be combined with one another, too, athough a few combinations are prohibited. Thus, for example, the “EBS” (or, equivalently, “ESB”) command means Extract-a-Specified-version-for-Browse.

Version Numbers

Both *major:minor* “trunk” version numbers and N-way branches are supported by TLIB 5.xx. The new, extended syntax for version numbers (which supports these features) is upwardly-compatible with the old TLIB 4.12 version numbers.

Minor version numbers

Some of our users who are used to other version control products have asked us to support “major:minor” version number pairs for trunk versions, rather than just integers. Thus, rather than having a sequence of “trunk” versions like “1, 2, 3, 4, 5, 6, 7”, you could have a series like “1, 1:1, 1:2, 1:3, 2, 3, 3:1”.

The numbers after the colons are called “minor version numbers,” and the “regular” numbers (before the colons) are called “major version numbers.”

Note that “1:0” and “1” are synonymous, so the second sequence of version numbers could also have been written, “1:0, 1:1, 1:2, 1:3, 2:0 3:0, 3:1”.

There isn't really any functional difference between the regular integer-only trunk version numbering scheme and the major:minor numbering scheme. Both are simply ways of designating a series of versions, each version derived from the one before.

Some programmers prefer to designate “minor” changes by incrementing the minor number, and “major” revisions by incrementing the major number. How -- or whether -- you use minor version numbers is entirely up to you. However, there are a few things to be aware of:

* Older versions of TLIB (e.g., TLIB 4.12) and Opus Make (prior to Opus Make 6.0), do not support major:minor version number pairs in TLIB libraries. If you use them, your TLIB libraries will be incompatible with the old releases of these programs.

* The use of major:minor version numbers does not affect retrieval of the "latest" trunk version. Versions "*" and "*:*" are synonymous; both mean "latest trunk version".

* The U and N commands work as in previous versions of TLIB: they create "regular" integer version numbers (that is, they increment the major version number and set the minor version number for the new version to zero). Thus, if you use the U command to add a new version after version 5:3, the new version will be version 6.

* You can use the US or UM command to force a new version to increment the minor number. With the US command, you can specify the specific version (e.g., "5:4"), or you can use an asterisk ("5:*"). More conveniently, however, you can use the M ("minor") suffix, which simply causes the U command to increment the minor version number instead of the major version number.

The M or S suffix can be combined with other suffixes (but not with each other). So, for example, to increment the minor version number and still keep the file checked-out/locked, use the UKM (or, equivalently, UMK) command. (The K suffix, to "Keep checked out," is explained on p. 101.)

* You cannot normally "skip" version numbers. That is, version 5:3 can be followed by either "5:4" or 6, but not by "5:5". However, this restriction can be circumvented via the `RELAXVERS` parameter, p. 307.

* You cannot normally have a version numbered zero. However, this restriction, too, can be circumvented via the `RELAXVERS` parameter, p. 307.

N-way branching

What this means is that you can have as many branch versions as you wish, all of which call the same version "Momma."

For example, suppose you have a standard version of module `XYZ.PRG` and it is currently at version level 5. You could make a custom modification of `XYZ.PRG` for one customer and call the new version "5.1".

But what happens when you make another customization of `XYZ.PRG`, for another customer? What do you call it? Version 6 is what you plan to call the next generic release, so you don't want to use that.

You need *another* version 5.1! In fact, if you make a lot of customized versions of `XYZ.PRG`, you might need *dozens* of different 5.1 versions, each of which begins a branch of its own in the "tree" of version numbers.

Of course, you need a way to differentiate the "first 5.1" from the "second 5.1" and "5.1 number three." So, we've expanded the version number syntax by adding an optional parenthesized "branch number," like this:

`5.(1)1`	- read aloud as "the first 5.1" or "5 dot branch 1 number 1"
`5.(2)1`	- read aloud as "the second 5.1" or "5 dot branch 2 number 1"
`5.(3)1`	- read aloud as "the third 5.1" or "5 dot branch 3 number 1"

Note that the old syntax is still allowed; "5.1" is just a shorter way of saying "5.(1)1", or "5.1 number 1":

`5.1`	- the same as "5.(1)1", "the first 5.1"

Also note that you can have branches from trunk versions even if the trunk versions are numbered with the new major:minor number pairs. For example:

`5:2.(4)1` - read aloud as "5 colon 2 dot branch 4 number 1", or maybe "the 1st version in the 4th branch from 5 colon 2".

Plus, you can have branches off of branches, off of branches, etc., as much as ten levels deep -- way more than you will ever need.

Complex version numbers can be kind of hard to read. Fortunately, if you don't need them, you don't have to use them. That's the beauty of this syntax: nearly everything is optional, and many users will never need anything except plain, integer version numbers.

Note that you can substitute an asterisk for the last branch version, to mean "latest." For instance:

`5:2.(4)*` - read aloud as "the latest version in the 4th branch from version 5 colon 2," or something like that.

Note: Older versions of TLIB (e.g., TLIB 4.12) and Opus Make (prior to Opus Make 6.0) do not support N-way branch version numbers and minor version numbers in TLIB libraries. If you use N-way branches, your TLIB libraries will be incompatible with the old releases of these programs.

See Appendix E for a more detailed description of TLIB 5.50 version number syntax, including a formal (BNF) grammar.

Specifying versions by version label

TLIB 5.50 supports several kinds of version label files. You use them interchangeably with version numbers to select versions, with the *@filename* syntax. For instance, if you had created the snapshot version label file `SNAPSHOT.X12` and wanted to use it to specify the version numbers of your `.c` and `.h` files, then the following command would extract all `.c` and `.h` files for which there are TLIB libraries, selecting the versions recorded in `SNAPSHOT.X12`:

```
        TLIB ES *.C,*.H @SNAPSHOT.X12
```

Specifying versions by date/time

As with earlier versions of TLIB, with TLIB 5.50 you can specify versions to be extracted by date and time, rather than by version number or version label.

However, TLIB 5 adds a restriction: If you specify a time-of-day in lieu of a version number to select a particular version with the "ES" command, you now must specify the time in hh:mm:ss format (not hh:mm format). That is, you cannot specify only hours and minutes; you must specify the seconds, as well.

This restriction was added to enable the TLIB "ES" command to correctly distinguish between the major:minor version number syntax and a time-of-day used to select a version. Thus, "1:12" is interpreted as a major:minor version number specification, and "1:12:00" is time-of-day.

Long File Names

The 32-bit versions of TLIB have full support for Windows long file names.

The real-mode DOS version of TLIB, `TLIBDOS.EXE`, does not support long file names.

Protected mode 16-bit versions of TLIB, such as `TLIB2.EXE` (for OS/2) have limited support for long file names: Paths and names are still limited to 80 characters total length, rather than 259, blanks are not permitted within file names, and you can't enclose filenames in quote marks.

However, a few of the minor auxiliary programs that come with TLIB do not support long file names.

Note: you can configure the `FNAMECASE` parameter to force mixed-case filenames to upper-case; see p. 309. Similarly, the `LONGNAMES` configuration parameter can be used to prevent TLIB from handling long file names:

```
 LONGNAMES <Y/N/M>
```

The three possible settings are:

`LONGNAMES Yes` Enables use of long file names. This is the default for most versions of TLIB.

`LONGNAMES No` Disables use of long file names by TLIB. This is the default for all versions of TLIB when running under DOS 6.xx and Windows 3.1 or 3.11.

`LONGNAMES Maybe` Enables use of long file names. For the DOS-extended command-line version of TLIB, `TLIBX.EXE`, and for the 16-bit GUI version of TLIB, this setting also enables automatic fallback to old-style (8.3) file access if long file name access fails. This is the default setting for those versions of TLIB when running under Windows-9x. For other versions of TLIB, `LONGNAMES M` is equivalent to `LONGNAMES Y`.

The `LONGNAMES` parameter is permitted but ignored by TLIB for DOS (`TLIBDOS.EXE`).

Here is a table summarizing long file name support by 16-bit versions of TLIB under various operating systems:

	longnames	MS-DOS	OS/2 native	OS/2's DosBox	Win 3.1x, Win-OS2	Win-9x, Win-Me	Win-NT, Win-2K	Win-XP
TLIB for Windows (API & GUI)	default	n/a	n/a	n/a	8.3	long	8.3	8.3
	N	n/a	n/a	n/a	8.3	8.3	8.3	8.3
	M	n/a	n/a	n/a	8.3*	long	8.3*	8.3*
	Y*	n/a	n/a	n/a	n/a	long*	n/a	n/a
TLIB2 (TLIB for OS/2)	default	8.3*	long	8.3*	8.3*	8.3*	long	8.3*
	N	8.3*	8.3	8.3*	8.3*	8.3*	8.3	8.3*
	M or Y	8.3*	long	8.3*	8.3*	8.3*	long	8.3*
TLIBX (DOS-extended)	default	8.3	n/a	8.3	8.3	long	8.3	8.3*
	N	8.3	n/a	8.3	8.3	8.3	8.3	8.3*
	M	8.3*	n/a	8.3*	8.3*	long	8.3*	8.3*
	Y*	n/a	n/a	n/a	n/a	long*	n/a	n/a
TLIBDOS	(any)	8.3	n/a	8.3	8.3	8.3	8.3*	8.3*

Legend:

8.3	supports short file names only
8.3*	supports short file names, but this is not the best version of TLIB to run in this operating environment, or it is not the recommended setting for `LONGNAMES` in this operating environment
long	supports long file names (80 chars maximum, no spaces)
long*	supports long file names, but we do not recommend configuring `LONGNAMES Y` under normal circumstances.
n/a	not supported

Note that you can use `IFF/ELSE/ENDIF` directives with the `%TLIBNAME%` symbol to set configuration parameters which apply only to certain versions of TLIB. For example, the following would disable support by TLIBX and TLIB for Windows for Win-95 long file names, without disabling OS/2 and NT long file name support in `TLIB2.EXE`:

```
iff ('%TLIBNAME%' eqi 'TLIBX') or ('%TLIBNAME%' eqi 'TLIBDLL')
   REM  Disable Win-95 long file names:
   longnames n
endif
iff '%TLIBNAME%' eqi 'TLIB2'
   REM  This is not needed, because it's the default:
   longnames y
endif
```

See also: `FNAMECASE` (p. 309), which determines whether file names are translated to upper-case (or lower-case).

Environment variables, SET, and the Autoset file

Referencing environment variables in your TLIB configuration file

You can reference DOS/Windows environment variables in your TLIB configuration file. Use the same syntax that you would use in a DOS `.bat` file or an OS/2 `.cmd` file: if you want TLIB to look up an environment variable called `NAME`, and insert the value of `NAME` at some point in your TLIB configuration file, you simply embed the string `%NAME%` in the configuration file at the point(s) where you wish the value to be inserted.

In other words, an environment variable (or other "set" name, as described below) can be used as a sort of macro to be inserted in your TLIB configuration file.

In accordance with the usual DOS convention, if `NAME` is undefined, then `%NAME%` is considered to be of zero length.

If you would prefer that an error message be generated for undefined names, then use the syntax, `%!NAME%`, instead. The "`!`" indicates to TLIB that it should display an error message if the name is undefined.

If having the name undefined would be catastrophic, then you may prefer to use the syntax, `%!!NAME%`, instead. The "`!!`" indicates to TLIB that it should display an error message and halt if the name is undefined.

Thus, you can chose to have an undefined name be considered equivalent to a 0-length value (no error), or it can generate a warning message, or it can generate a "fatal" error (which prevents TLIB from running).

Also, you can now optionally use parenthesis to enclose all or part of the set variable name.

The `!` or `!!` prefix determines what happens if the variable name is undefined. If the variable name is defined, the `!` or `!!` prefix has no effect.

```
%NAME%   or  %(NAME)%        - if NAME is undefined, it's okay (0-length)
%!NAME%   or  %!(NAME)%      - if NAME is undefined, it causes a warning
%!!NAME%   or  %!!(NAME)%  - if NAME is undefined, TLIB will not run
```

To use an actual, literal percent sign in the configuration file where it would otherwise appear to be part of something of the form `%NAME%`, you can double the percent sign.

For example, if your `AUTOEXEC.BAT` or `STARTUP.CMD` contained the line

```
  SET TMP=F:\
```

you could add to your TLIB configuration file the line

```
  help 1,"TLIB -- %%=percent sign, "%TMP%"=temp directory"
```

and then the help screen in command-line versions of TLIB would display

```
 TLIB -- %=percent sign, F:\=temp directory
```

Note: Referencing environment variables in your configuration file currently works *only* with TLIB itself (both command-line and GUI versions). It does not work with the other programs in the TLIB package which read the TLIB configuration file (`CMPR` and `TLMERGE`).

Parenthesis

The parenthesized forms allow you to dereference more than one level deep. For instance, suppose you had the following SET variables:

```
  set X=BBB
  set BBBMSG=Hello
```

Then you could configure

```
  help  1,'X=%X%, and BBBMSG contains %(%X%MSG)%'
```

and the first line of the command-line TLIB help screen would be

```
 X=BBB, and BBBMSG contains Hello
```

Obviously, that example is a bit contrived. However, we're confident that someone, somewhere will find a use for this sort of thing.

Choosing defaults to be overridden by environment variables

We've subtly changed the way TLIB parses its configuration file, so that you can now configure TLIB to let an environment variable (or other "set" name) override almost any configuration parameter.

The change is simply to stop parsing the line at the first blank or tab after the configuration parameter argument (except for a few configuration parameters, mostly those which allow embedded white-space in the argument).

This allows you to configure, for example:

```
  DEFEXT %newdefext% PAS
```

If `NEWDEFEXT` is not defined, then the line is equivalent to

```
  DEFEXT  PAS
```

(Note that the extra space before "`PAS`" doesn't matter.)

However, if you define `SET NEWDEFEXT=C` then the line is equivalent to

```
 DEFEXT C PAS
```

which would have been an error in TLIB 4.12, but is now equivalent to

```
  DEFEXT C
```

Thus, you've effectively configured TLIB is such a way that `DEFEXT` is `PAS` except when you define an environment variable (or other set-name) called `NEWDEFEXT` to override it.

Here's another example:

```
  LOCKING %LOCKS% Y
```

This configures `LOCKING` to the value of environment variable `LOCKS` (which should be either `Y`, `N`, `Weak`, or `Branch`), but if `LOCKS` is not defined then `LOCKING` is configured to `Y`.

Note that this technique doesn't work for the following configuration parameters:

```
 ARCCMD COMMANDS EXTENSION HELP IF KEYFLAG LOGFLAG LOGPREFIX
```

`LOGSUFFIX PATH PROMPT SET UNARCCMD`

“SET” parameters in your configuration file

You can use `SET` configuration parameters to define pseudo-environment variabfles, which you can reference via the `%NAME%` (or `%!NAME%` or `%!!NAME%`) syntax, just like real environment variables. The syntax is just like the DOS or OS/2 “set” command:

```
  SET NAME=string
```

The `SET` configuration parameter overrides any normal environment variable setting for the same name.

The autoset file

You can create a file in the current directory called `AUTOSET.BAT` (under DOS or OS/2's DOS box), or `AUTOSET.CMD` (in OS/2 protected mode), which contains more pseudo-environment variable definitions. This file is called the “autoset file.”

The first time TLIB encounters a `%NAME%` reference in a configuration parameter, it will look for the autoset file, and if one is found TLIB will read it. The autoset file should contain “`SET NAME=`*unquoted-string*” commands (in the usual DOS & OS/2 format). These commands override any normal environment variable settings.

This gives you a way to have something resembling “local” environment variables: names for which the definition depends upon your current directory. By referencing such names in your TLIB configuration file, you can make TLIB's behavior depend upon your current directory, even if you use a single TLIB configuration file regardless of which directory you are working in (perhaps by setting your `TLIBCFG` environment variable).

Note that the autoset file does not actually cause changes to your DOS or OS/2 environment settings (unless you run it as a batch file). The autoset file only affects `%NAME%` references embedded in TLIB configuration files (and in configuration lines specified via the C command). In particular, the autoset file does not affect the use of the `TLIBCFG` environment variable to specify the location of the configuration file.

Note: to make `TLIB2.EXE` read `AUTOSET.BAT` instead of `AUTOSET.CMD`, see the `AUTOSET` configuration parameter, p. 292.

Precedence of different kinds of SET names

The `SET` configuration parameter works like an environment variable or a “set” command in the autoset file. The syntax is the same, too:

```
SET name=something
```

Like set commands in the autoset file, the `SET` configuration parameter does not really alter DOS (or OS/2) environment variables, but that distinction is mostly academic, since set names can be used exactly like environment variables in TLIB.

There are three ways to define set names:

- With a DOS or OS/2 environment variable

- With a “set” parameter in the TLIB configuration file

- With a “set” command in the autoset file

If the same name is defined in more than one way, the following precedence rules are used:

o Environment variables have the lowest precedence. They can be overridden by `SET` commands in either the TLIB configuration file or the autoset file.

o `SET` parameters at the beginning of the TLIB configuration file (before the first `%NAME%` reference in the configuration file) have lower precedence than `SET` commands in the autoset file.

o `SET` parameters at the end of the TLIB configuration file (after the first `%NAME%` reference) have higher precedence than `SET` commands in the autoset file.

In other words, the autoset file's `SET` commands are effectively inserted into the configuration file just before the first `%NAME%` reference.

Predefined pseudo-environment variables

TLIB 5.50 has over 50 predefined special `SET` variables of the form "`TLIBCFG:`*name*".

Most of the TLIB configuration parameter values can be referenced in this way. For instance, `%TLIBCFG:projlev%` is the value of the `PROJLEV` configuration parameter (name of the current project level). Similarly, `%TLIBCFG:id%` is the current user ID.

One use of this feature is to allow you to display configuration parameters in the prompt, help or banner.

Restrictions:

A) The following configuration parameters do not have `%TLIBCFG:`*name*`%` variables defined for them:

```
AATTR, BANNER, D3COLLIDE, D3FLAG2, D3FLAG3, ENDIF, HELP, IF,
INCLUDE, JOPTIONS, KEYFLAG, LOGFLAG, LOGPREFIX, LOGSUFFIX,
PROMPT, REM, SET
```

B) You should not reference `%TLIBCFG:`*name*`%` if configuration parameter "name" is defined in an `IF`/`ENDIF` block. Doing so will not cause an error, but the behavior may change in future editions of TLIB Version Control.

You can also get the path\name of the TLIB configuration file(s), as if they were environment variables, from within the configuration file(s), via the `%tlibcfg:...%` syntax. Two forms are allowed: `%tlibcfg:cfgfile%` and `%tlibcfg:curfile%`.

For example, to display the path\name of the main configuration file whenever TLIB starts, you could configure:

```
  numbanner 1
  banner 1,"config file='%TLIBCFG:CFGFILE%'"
```

Even if you are using include directives, `%TLIBCFG:CFGFILE%` is the name of the first/main configuration file, not the included configuration file. To get the name of the current configuration file, use `%TLIBCFG:CURFILE%`. For example, in a "master" `tlib.cfg` intended to be included from other `tlib.cfg` files, you might configure this:

```
  numbanner 2
  banner 1,"main/first config file='%TLIBCFG:CFGFILE%'"
  set masterfile=%TLIBCFG:curfile%
  banner 2,"master config file='%MASTERFILE%'"
```

Note the use of a temporary `SET` variable to store a copy of `%tlibcfg:curfile%`. It is needed because by the time that TLIB displays the `BANNER`, it is no longer reading configuration files, so there is no "current" configuration file.

Additionally, the following predefined pseudo-environment variables can be referenced within TLIB configuration files:

```
  %TLIBOS%          (may be: DOS, OS2, or WINDOWS)
  %TLIBMODE%        (may be: REAL or PROT)
  %TLIBNAME%        (may be: TLIB, TLIBX, TLIB2, or TLIBDLL)
  %TLIBPROG%        (may be: EXE or DLL)
  %TLIBWORDSIZE%    (may be: 16 or 32)
```

Example (try adding these 2 lines to your tlib.cfg file):

```
 Say %TLIBNAME%.%TLIBPROG% is running
 Say in %TLIBMODE% mode (%TLIBWORDSIZE%-bit).
```

Example:

Suppose `TLIB.CFG` contains:

```
  PROJLEV %CurLevel%
  LEVEL n=test d=D:\TEST\ p=release
  LEVEL n=release d=D:\RELEASE\
               prompt  1,'Hi, %TLIBCFG:id%! Current level = %
TLIBCFG:projlev%'
```

And `AUTOSET.BAT` (or `AUTOSET.CMD` for `TLIB2.EXE` under OS/2 or NT) contains:

```
  set curlevel=test
  set tlibid=Dave
```

then the first TLIB prompt line would be:

```
 Hi, Dave!  Current level = TEST
```

Note: `%NAME%` and `%TLIBCFG:`*name*`%` references in the prompt, help and banner screens are evaluated "late," so that they always reflect the current values (rather than the values in effect when the configuration file was read). In all other contexts, however, such references are evaluated immediately, as the TLIB configuration file is read.

TLIB also supports an extension to this syntax for referencing the current work directory path, which allows you to use pieces of your work directory path at other places in your TLIB configuration file, in other configuration parameters.

Thus, for example, if you have a large number of projects, you can configure TLIB with a "generic" project level that is named the same as whatever your current work directory is.

The syntax is an extension of the `%TLIBCFG:WORKDIR%` construct which lets you specify one or two numbers to select which parts of the path you want.

Suppose that `%TLIBCFG:WORKDIR%` is "`C:\WORK\ABOMB\`". This path is considered to have three parts: part #0 is the drive/root ("`C:\`"), part #1 is the top subdirectory ("`WORK`"), and part #2 is the lower subdirectory ("`ABOMB`").

The syntax to specify specific parts of the path is: `%TLIBCFG:WORKDIR:`*nn*`:`*mm*`%`, where *nn* is the number of the left-most part, and *mm* is the number of the right-most part. (Also, if *mm*=*nn*, that is if you only want one part of the path, you can leave out the "`:`*mm*".)

For example, if `%TLIBCFG:WORKDIR%` is "`C:\WORK\ABOMB\`" then:

```
  %TLIBCFG:WORKDIR:0:0% = "C:\"
  %TLIBCFG:WORKDIR:0%   = "C:\"
  %TLIBCFG:WORKDIR:1%   = "WORK"
  %TLIBCFG:WORKDIR:2%   = "ABOMB"
  %TLIBCFG:WORKDIR:1:2% = "WORK\ABOMB"
  %TLIBCFG:WORKDIR:0:1% = "C:\WORK"
  %TLIBCFG:WORKDIR:0:2% = "C:\WORK\ABOMB"
  %TLIBCFG:WORKDIR%     = "C:\WORK\ABOMB\"
```

You can also count the parts from the right-most, by specifying negative numbers. The right-most (lowest level) directory is part `-1`. This is useful when you want the current directory name, but don't know how "deep" it is.

So for the example above, part `2` can also be specified as part `-1`, and part `1` as part `-2`:

```
  %TLIBCFG:WORKDIR:-2%   = "WORK"
  %TLIBCFG:WORKDIR:-1%   = "ABOMB"
```

To experiment with this feature, use `SAY` or `BANNER` or `PROMPT` to display the results, like this:

```
  set xxx=%tlibcfg:workdir:2:-1%
  set yyy=%tlibcfg:workdir:-1%
  say Note: xxx=‘%xxx%’  yyy=‘%yyy%’
```

Then run TLIB to see what values `xxx` and `yyy` get. (In TLIB for Windows, click Run or View Log to see messages in the status log.)

Warning: Be sure that if you configure either `TREEDIRS Y` or `WORKDIR`, you do so *before* referencing `%tlibcfg:workdir%`, to ensure that TLIB deduces the correct work directory!

Here's an example of how you might use this feature to set up a generic project level, whose name was deduced from the name of your current work directory:

```
 set dir=%tlibcfg:workdir:-1%
 level n=%dir% d=\\server\sys\levels\%dir% a=Y
 projlev %dir%
 ! createtf y  <-- this is optional
```

Here's a similar, but fancier configuration, which defines a generic ISO-9001-style 3-level “promote” structure:

```
 set dr=%tlibcfg:workdir:-1%
 level n=%dr%_DEV d=x:\levs\%dr%\dev\ a=y f=y p=%dr%_TST i=%dr%_REL
 level n=%dr%_TST d=x:\levs\%dr%\tst\ w=n f=y p=%dr%_REL
 level n=%dr%_REL d=x:\levs\%dr%\rel\ w=n
 ! Developers set projlev like this:
 projlev %dr%_DEV
 ! Testers should configure:
 ! projlev %dr%_TST
```

Lots of ways to set your TLIB User ID

TLIB supports several different ways of setting the TLIB User ID. You can set the `ID` configuration parameter in your TLIB configuration file, or you can use the CW ("configure who") command on the TLIB command line with command-line versions of TLIB, or in TLIB for Windows you can set a User ID override which will be stored in the `TLIB.INI` file.

The `ID` configuration parameter defaults to `%TLIBID%`, so that the `TLIBID` environment variable can also be used to set your user ID (e.g., in your autoexec.bat file, "`SET TLIBID=DAVE`"). In the event that you use more than one method to sent your user ID, the order of precedence is:

[1] In TLIB for Windows, the `TLIB.INI` override, set via `File -> Configuration Options -> Set User ID.` (highest priority)
[2] The CW ("configure who") command in command-line TLIBs
[3] The `ID` configuration parameter
[4] The `TLIBID` environment variable (lowest priority)

Note: In TLIB 4.12 and earlier, the order of precedence was different:

[1] The W ("who") command (highest priority)
[2] The `TLIBID` environment variable
[3] The `ID` configuration parameter (lowest priority)

Network name look-up

Under Windows-NT, you can configure `ID %USERNAME%` to set TLIB's user ID to the usual environment variable.

For DOS, Windows, and OS/2, TLIB may be able to interrogate your network software to find your network login ID, depending upon what kind of network you have. (We are very grateful to Mr. Mark Evans for extensive help implementing this feature.)

There are several "special" TLIB user ID's which you can set, each of which will be translated in a different way by looking up the actual name which you are using on your network. The special names are:

`*NETNAME*`	Looks up the NETBIOS login ID (under MS-DOS and Windows), or the LAN Manager login ID (under OS/2).
`*NOVELL*`	Looks up the Novell login ID.
`*NOVELL2*`	Looks up the Novell login ID (alternate method).
`*MACHINE*`	Looks up the "machine name."
`*LANTASTIC*`	Looks up the Artisoft Lantastic login ID, but only under DOS and Windows. One caveat: if you are logged into two different servers with two different user names, there currently is no way to predict which user name will be returned.
`%USERNAME%`	Use the Windows-NT/2K/XP `USERNAME` environment variable.

To make TLIB look up your network user ID, just use the `ID` configuration parameter (or the `TLIBID` environment variable, or the "`CW`" command) to specify one of the special `*name*` names as your TLIB user ID.

Because of the great variety of networking software in use today, there is no guarantee that any of these special user IDs will necessarily work with your network. However, trying them out is easy. Just run the command-line version of TLIB interactively, select the "CW" (configure who) command, and enter each of the four special names, in turn, and see what happens. In TLIB for Windows, just add an experimental `ID` configuration parameter to the end of `TLIB.CFG`, or you can use Run manual command. Note that if you are using OS/2, you should do this test from a DOS box as well as at the regular OS/2 command prompt. Example:

```
  TLIB CW *LANTASTIC*
```

Or configure:

```
  ID *LANTASTIC*
```

Also, note that we currently use two different approaches to implementing the DOS real mode `*NOVELL*` name look-up.

Additionally, TLIB can store a user ID override in its `C:\TLIB.INI` file. To use this feature, in TLIB for Windows choose `File` -> `Change User ID` and the “`Always`” override button; see p. 251.

S command: Snapshot version labeling

An important problem for developers using many source files is coordinating them all. Consider what happens when you need to reconstruct an old version of a program. If it is made with just one source file, you simply run TLIB and retrieve the desired file. But if there are 20 different source files, you are faced with the prospect of manually selecting the proper version from each of 20 different library files -- a tedious and error-prone process.

To solve this problem, TLIB provides the S command (which replaces the old TLIBSNAP utility). The S command takes a "snapshot" of the current state of each of the library files for a set of related source code files, so that later you can easily reconstruct the version. You would typically use the S command whenever you release a new version of your program. The S command will create a "snapshot file" (sometimes called a "version label file" or "class"), which is a kind of file list that contains version numbers as well as file names.

Command line syntax:

`TLIB S` *snapshot-file wild-cards-or-@filelist*
`TLIB SS` *snapshot-file wild-cards-or-@filelist version-spec*

Using the snapshot to retrieve old versions

A snapshot created by the S command is simply a file which records a "snapshot" of what source files you are using and what their current version numbers are at a particular time. It is a text file containing file names and version numbers.

If you ever need to reconstruct that version of your program, simply retrieve the snapshot file and use it as a file list, as input to the TLIB E (extract) command. TLIB can use a snapshot as a special form of file list which specifies both file names and version numbers. All the proper source files will be automatically retrieved. For example, if you have a snapshot named `beta.snp`, then the following command would retrieve the recorded versions of all the files listed within it:

```
  TLIB E @beta.snp
```

Alternately, you can use a snapshot to specify just version numbers, selecting the files in which you are interested in some other way. For instance, if you had a snapshot called `beta.snp`, for the "beta test" release of your program, the following command would extract `myfile.c` and `myfile.h` from their TLIB library files, taking the version numbers from `beta.snp`:

```
  TLIB ES myfile.c,myfile.h @beta.snp
```

The TLIB ES command requires two parameters: (1) the file names that you want to retrieve, and (2) the version numbers. In the following example, both are specified as `@beta.snp`.

```
  TLIB ES @beta.snp @beta.snp
```

Thus, these two commands do the same thing:

```
  TLIB E @beta.snp                  this is the fast way
  TLIB ES @beta.snp @beta.snp       this way is slower
```

What version numbers are recorded

The S (snapshot) command records the "current" version numbers for your source files. That is, the S command records the version numbers which the E command would extract.

In the simplest case, when **projlev** is not configured, the current version numbers are simply the latest (highest) "trunk" version numbers.

If you are using named project levels (p. 139), the current version numbers are those that are currently listed in the project level tracking file for the current project level (or perhaps in a "parent" project level).

A special case is when "`projlev =`" (or, equivalently, "`projlev equals`") is configured. This is used occasionally for special purposes, but isn't recommended for most users. With this configuration setting, there is no named project level, and the current version numbers are those recorded in the work directory tracking file.

In summary, the S command records the latest trunk versions unless `PROJLEV` is configured to a named project level (in which case the versions which are current in that project level are recorded), or unless `PROJLEV =` is configured (in which case the version nubers for tracked files are taken from the work directory's `tlibwork.trk` file).

Creating a snapshot

Note: If you are using project level tracking (pp. 299 , 139 & 143) with fully-populated project levels (also explained later), you may have little use for the S command. Simply saving a copy of your project level tracking file will do the trick, since it contains all the module names and version numbers that you are using.

However, if you are not using project level tracking, or if your project levels are “sparse” (that is, they contain only those files that differ from a parent level), then you'll need to use the S (snapshot) command to make version labels.

Example 1: Snapshot the current versions of all files defined in all active project levels (the current level and its parent levels):

```
  tlib sa snap.xxx *.*
```

Explanation: The “A” search mode suffix causes TLIB to examine the promote-to and/or inherit-from level(s) as well as the current level. The “`*.*`” wild-card spec says that we want to record all of the files. `Snap.xxx` is the output file.

Example 2: Snapshot all library files, recording the latest trunk versions, without regard to your project levels.

```
  tlib ss snap.xxx *.* *
```

Explanation: The default search mode (“L”, for library files) is used, since no wild-card search mode is specified. The “`*.*`” file specification indicates that all library files are to be searched. The “S” suffix indicates that you wish to specify the version numbers, and the version number specification given was “`*`” (which means latest trunk versions).

Note #1: You can use this form even if you are not using project level tracking.

Note #2: Like the old TLIBSNAP program, this form requires that TLIB read all the TLIB library files in order to determine what the latest trunk version is for each file. Thus, it may take several minutes if you have a large number of TLIB library files.

Example 3: Add additional source files to an existing snapshot:

```
  tlib s snap.xxx @snap.xxx,myfile.*,@extras.lis
```

Explanation: In this example, `snap.xxx` is presumed to be an existing snapshot file, to which you wish to add both `myfile.*` and all the files listed in `extras.lis`.

TLIB processes your three input file specifications in the order given, and it contains logic to ensure that no name will be processed more than once, so if any of the additional files were already recorded in `snap.xxx`, they will remain unchanged. (The additional files are `myfile.*` and the files listed in `extras.lis`.)

To make this case work, TLIB detects the fact that the output snapshot file, `snap.xxx`, is also used as an input file. To avoid clobbering it, TLIB writes the output to a temporary file, and when done processing all of your input files, TLIB will copy the temporary file to `snap.xxx`.

Note #1: This ability to list the output snapshot file as an input file, as well, obviates the need for the old TLIBSNAP "A" (append) command.

Note #2: TLIB looks for your `TMP` (or, as a 2nd choice, `TEMP`) environment variable to determine where the temporary file should be created. The temporary file is usually named "`$TLIB_TM.2`", but you'll probably never see it, since TLIB deletes it when done.

Example 4: Alter an existing snapshot, changing old version numbers, and/or adding additional files:

```
  tlib s snap.xxx myfile.*,@extras.lis,@snap.xxx
```

Explanation: This example is just like the last one, except that `@snap.xxx` is given as the last input file specification rather than the first. This has the effect of using the current version numbers for `myfile.*` and the files in `extras.lis`, rather than the old version numbers that were already given in `snap.xxx`.

Note #1: You will probably want to put your snapshot files under version control, by storing them in a TLIB library file. This will allow you to store comments with each snapshot to record the purpose of the release, and it will avoid the necessity to keep inventing new names for your snapshot files. The TLIB for Windows GUI interface makes this convenient by offering the "Immediate Store in Library" check-box.

Note #2: In command-line versions of TLIB, the S command normally displays one dot (period) or comma for each source file that it finds, as an "activity indicator." If TLIB can determine the version number without reading the TLIB library (e.g., because the version number is found in a project level tracking file), then a dot is displayed. If TLIB must read the library file, then a comma is displayed.

If you wish to suppress these periods and commas (as well as several other "noisy" TLIB messages), you may configure `QUIET Y`, or run TLIB with the `-q` option as the first command-line parameter.

Check-In/Out locking: concurrent access control

This feature is for use when more than one programmer may be working on a program. It is disabled unless you specifically enable it in the TLIB configuration file. So, if yours is a one programmer shop, you needn't bother to read this chapter.

If you enable “locking” in your configuration file, then TLIB will create, modify and and examine “lock” files to control which programmer currently “owns” a library, to ensure that two or more programmers are not working on the same file at one time.

The Purpose of Check-in/out Locking

Check-in/out concurrency control (“locking”) is designed to solve a problem faced by developers who share responsibilities for a large programming project. The problem is that sometimes two different programmers may decide to change the same source file at the same time. If neither one knows that the other is working on that file, then the result will be that each of them has a version which is down-level (out of date) in some respect. Then, after they each update the library file (with the TLIB U command), the newest version in the library file will be missing some improvements which are present in the second-newest version, and vice-versa.

At best, the problem will be discovered early, and the programmers will have to reconcile their differing versions (manually, or with DIFF3 or TLIB's Migrate command). At worst, a “fixed” bug will be “unfixed” in the latest version, with potentially expensive consequences.

The solution is simple: just make sure that only one programmer is working on a module at a time. A manual procedure to ensure this would involve keeping a notebook for “checking out” each source file. Before extracting a source file from its library file to make modifications, a programmer would first “sign it out” in the notebook. If someone else was in the process of changing the file, the programmer would see in the notebook that the file was currently “checked-out” to the other person, and would know not to change the file until the other person had finished with it and “checked it in” again.

TLIB automates this process. Instead of a notebook for “signing out” each file, TLIB keeps a “lock” file. The lock file contains the name of the programmer who is currently working on the corresponding source file. When no one has a module signed out, then the lock file does not exist, and the library is said to be “unlocked”.

The lock file is similar in purpose to a Unix SCCS “p-file.”

Weak Locking and Branch/Level Locking

TLIB supports two regular locking modes, plus two special modes. The regular modes are:

```
 LOCKING N  “None” (disabled -- no locking)
 LOCKING Y  “Yes” (full, whole-library locking)
```

With `LOCKING N` configured, TLIB doesn't track who is working on a module, and doesn't restrict the storage of new versions. In this mode, the “E” and “EB” commands are equivalent, as are the “U” and “UK” commands.

`LOCKING N` is intended for use by individual programmers, working alone.

With `LOCKING Y` configured, TLIB allows just one programmer to work on any particular module. If a second programmer tries to extract (checkout) the module for modification (the “E” command), he will be greeted with an error message which tells him that the first programmer is already working on the module.

`LOCKING Y` is suitable for most multi-programmer development projects which do not utilize multiple teams working concurrently on different levels of code at the same time.

TLIB's two special locking modes, `LOCKING B` and `LOCKING W`, allow for concurrent development on a single module by multiple programmers.

The special concurrent-development modes are:

`LOCKING B`	"Branch" or "level" locking. Check-out locks only the current project level, although warnings will be generated when another programmer checks out the module from another project level. (Project levels are explained later.)
`LOCKING W`	"Weak" or "warning-only" locking. Check-out is allowed by any number of programmers simultaneously, even on the same project level, but a warning is displayed if you check-out a module that someone else is already working on.

In order of "strength" of locking, the four modes are:

`LOCKING N`	*(weakest)*
`LOCKING W`	
`LOCKING B`	
`LOCKING Y`	*(strongest)*

If you intend to use the AP command (in TLIB for Windows, `Add/Alt -> Promote`) to "promote" modules to "higher" assurance levels, then you should use `LOCKING B` instead of `LOCKING Y`, so that having a module checked-out at a "lower" level won't interfere with promoting it to the higher level. (The AP command and "promote levels" are used for "staged development" on large, multiple-programmer projects; see "promote" in the index.)

Warning: `LOCKING B` (branch or level locking) should only be used with named project levels for which "`s=New`" or "`p=`*something*" has been configured in the `LEVEL` configuration parameter, or for which there is no "`i=`*something*" linkage, since only the current project level will be locked. (These fields of the `LEVEL` configuration parameter are explained under "Configuring Your Project Levels," p. 141.)

Note that programmers can mix and match locking levels by changing the `LOCKING` configuration parameter, either with `IF`/`ENDIF` blocks in the configuration file or with TLIB's C (configure) command. If a programmer locks a module with `LOCKING Y` configured, then all other programmers will be denied modification-access (unless they configure `LOCKING N`). However, if the module is locked with `LOCKING B` or `LOCKING W` configured, other programmers will still be allowed to check-out the module for modification (though for `LOCKING B` it will only be allowed on a different project level).

CW command: Configure Who you are (set User ID)

Before he can check a file in or out, a programmer must give his name ("user ID") to TLIB via the `ID` configuration parameter or the `CW` (configure who) command; see p. 89.

(Note: the user ID should not contain blanks.)

E and U commands revisited

The `locking` parameter changes the behavior of the E (extract) and U (update) commands. When `locking y` is configured, the E command causes the file to be "checked-out" for modification by the current user; if the file was already checked-out to someone else, the E command will fail. When locking is enabled, the U command will cause the file to be "checked back in" when the library is updated with the new version; the file must have been already checked-out to the current user, otherwise the U command will fail. Note that even if the U command didn't update the library because there were no changes to the source file, the file will still be checked-in/unlocked (but see the `FORCEU` parameter, p. 275 , if you want to store a new version even when there are no changes).

If you enable or disable locking, you may also want to use the `PROMPT`, `HELP` and `COMMANDS` configuration parameters to customize the user interface for command-line versions of TLIB; see p. 329.

When locking is enabled (`LOCKING Y` is configured), TLIB's E (extract) command is similar to the Unix SCCS "`get -e`" command, the RCS "`co -l`" command, and the IBM PDL "`lget`" command; and TLIB's U (update) command is similar to the Unix SCCS "`delta`" command, the RCS "`ci`" command, and the IBM PDL "`lput`" command.

EB command: Extract for Browse

Sometimes you may wish to extract a source file from the library file without checking it out. If you do not intend to modify the source file, there is no need for the library to be locked. To extract the latest version without checking it out, the EB (extract-for-Browse) command is available. It works just like the E command, except that the lock file is ignored.

Note: The EB (browse mode) commands will never replace a source file which you already have checked-out for modification, regardless of how you have set the `REPLACE` configuration parameter. It will, however, replace a browse-mode copy of a source file, if you've configured `REPLROBR Y` (see p. 282 for details).

When locking is disabled the E and EB commands are exactly equivalent.

UK command: Update and Keep locked

Less often, you may wish to add a version of your source file to the library file without checking it back in. You might do this if you had finished one of several changes to a source file, and were ready to start working on the next change. To do this, you can use the UK (update-but-Keep-checked-out) command. The UK command works just like the U command, except that the lock file is not altered (it is still examined to ensure that you have already checked-out the file).

Similarly, the NK command works just like the N command, to create a new library file, storing the first version of your source file, except that the NK command leaves the source file checked-out/locked so that you can make additional changes to it.

When locking is disabled the U and UK commands are exactly equivalent.

With locking enabled, TLIB's EB (extract-for-Browse) command is similar to the Unix SCCS “`get`” command and the RCS “`co`” command, and TLIB's UK (update-but-Keep locked) command is similar to the RCS “`ci -l`” command.

UD (discard changes) and ER (reserve) commands

If you check-out a source file, intending to modify it, but later change your mind, you can use the UD (check-in, discarding changes) command to check it back in (unlock it) without updating the library file. As with the U command, if you mistakenly attempt to check-in a file which you do not have checked-out, TLIB will display an error message.

It is also possible (though rarely necessary) to check-out (lock) a file without actually extracting it from the library file, with the ER (reserve) command. As with the E command, if the file cannot be checked-out because someone else already has it, TLIB will display an error message telling who has the file checked-out.

Note that the library file need not exist for you to use the ER command to “reserve” it. Thus, you can use the ER command to reserve/lock a source file name before creating it, to ensure that nobody else can create a source file with that name. When you finally get around to creating the TLIB library with the N command, the source file will be checked-in/unlocked just like with the U command.

With locking enabled, TLIB's UD (check-in/discard changes) command is similar to the RCS “`rcs -u`” command and the IBM PDL “`lusing (release`” command, and TLIB's ER (check-out/reserve) command is similar to the RCS “`rcs -l`” command and the IBM PDL “`lusing (reserve`” command.

Note that the UD and ER commands are seldom used.

If locking is disabled, the E (extract) and EB (extract for browse) commands are equivalent, as are the U and UK commands, and the UD and ER commands have no function.

T command: Test lock status

The T (test) command can be used to test the check-in/out (lock) status of library files. If used interactively, the T command just displays informative messages indicating who has each file checked-out. If the T command is specified using DOS command line parameters (typically in a `.BAT` file), the DOS errorlevel is set as follows:

0 - checked-out to current user id
1 - not checked-out to anyone
2 - checked-out to someone else

As with all TLIB commands, if multiple files are specified (with wild-cards or a file list), then the errorlevel returned is the highest of the error-levels which would have been obtained by testing each file individually.

To test check-in/out status, the T command examines lock files, not library files. However, it will display a warning message if the corresponding library file does not exist (except when using PKZIP-compressed/archived library files).

By using wild-cards or file lists to specify a group of source files or library files, you can use the T command to produce a simple report listing the check-in/out lock status of each file. To store the report in a file (or print it), use DOS output redirection. For example:

```
  tlib t c:\libs\*.* >report.lst
```

Note: To test in a batch file whether *anyone* (including you) has any files checked-out, you can use an "`if exist`" test to check for the existence of the lock files.

Examples

Example #1: Suppose a programmer named "Dave" has his TLIB library files on LAN-shared disk drive `G:` in subdirectory `\TLIB`. He needs to change a source file called `MYFILE.C`. He can check-out and extract `MYFILE.C` from its library file like this:

```
  tlib cw Dave e myfile.c
```

After he is done making his changes, he could add the new version to the library file, checking it back in like this:

```
  tlib cw Dave u myfile.c
```

Note that the "CW" command can be omitted if the "`id`" parameter is specified in the TLIB configuration file (or if the "`set tlibid=Dave`" DOS or OS/2 command has been issued).

Example #2: A programmer named Jane needs a copy of `MYFILE.OBJ`, to be linked with some other files. She wishes to extract a copy of the source file, `MYFILE.C`, which she will compile to produce `MYFILE.OBJ`. Since she does not intend to change `MYFILE.C`, she should use the EB command to extract it, to avoid locking it:

```
  tlib eb myfile.c
```

If she had accidentally extracted `MYFILE.C` with the E command, like this...

```
  tlib cw Jane e myfile.c
```

...then she could check it back in with the UD (discard changes) command:

```
  tlib cw Jane ud myfile.c
```

In other words, the EB command is equivalent to an E command followed by a UD command.

Also, the UK (update-and-Keep-locked) command is equivalent to U (update) followed by E (check-out). Conversely, the U command is equivalent to UK followed by UD.

Related configuration parameters are:

`LOCKING`, `READONLYB`, `REPLROBR`, `DELETESRC`, `LOGUSER`, and `ID`.

Novell Netware users should also configure:

```
  DATAPATH Y
```

Security

There is now a semi-secret “patch point” which can be used to force TLIB to look in one and only one place for its configuration file. This is designed to add to the control available to you if you are a “system librarian” who is administering a project with a large number of programmers. By using this mechanism, you can prevent users from casually changing their configuration parameters.

If you wish to use this feature, you may want to:

A) Disable the “C” command via the `COMMANDS` configuration parameter.

B) Use the `%`*name*`%` syntax to enable users to set certain, specific configuration parameters.

C) Call us to find out the patch procedure (it is *not* in the manual).

With the configuration file's path “hard-wired” to a network directory, to which you can control write-permission, you can prevent users from changing their TLIB configuration files. By using this in combination with judicious use of environment variable references in the configuration file, you can allow users to change only certain configuration parameters, and not others.

Because TLIB's user interface is configurable, the ability to restrict configurability also provides you with the ability to restrict access to unwanted commands.

In addition, you can restrict access to the TLIB library and lock files by distributing them among several directories on the network file server, and granting users read/write access or read-only access or no access to the several directories based upon their needs.

See also pp. 244 and 260.

Version tracking & Named Project Levels

“Basic version tracking” is for automatic branching and automatic version labeling on very small projects. It is intended for single-programmer projects which do not utilize check-in/out locking.

What makes this form of version tracking “basic” is that `PROJLEV` is not configured at all (or is configured to one of the special pseudo-names, “`*`” or “=”). Because there are no named project levels, TLIB has to manage only one tracking file (at a time).

For more complex development environments, “advanced version tracking” (with named project levels) provides additional features and greater flexibility; it is explained later.

If you configure `TRACK Y` then TLIB will maintain (in your working directory) a “version tracking file” called `TLIBWORK.TRK`. It will contain the full list of those files in the directory that you have extracted from or updated into a TLIB library, the version number for each module, and sometimes some additional information about some of the modules.

The information in `TLIBWORK.TRK` is similar to the “snapshot version label file” created by the S command (or the old-format snapshot file generated under TLIB 4.12 by the TLIBSNAP program), but the file format is different.

Like a snapshot file, `TLIBWORK.TRK` contains a full list of module names and version numbers, so you can use it like you would use a snapshot version label file. To “take a snapshot,” you can use the S (snapshot) command, or you can simply save a copy of `TLIBWORK.TRK`, perhaps by using TLIB's U (update) command to store the latest `TLIBWORK.TRK` in its own TLIB library:

```
  TLIB U TLIBWORK.TRK This version fixes problem number IR759
```

Unlike TLIBSNAP, however, in its snapshot files and in the `TLIBWORK.TRK` tracking file TLIB 5.50:

A) Does *not* store the file name or path of the associated TLIB library file. This simplifies things if you need to move your TLIB library files around, but it means that you are responsible for using consistent `PATH` and `LIBEXT` configuration settings to ensure that TLIB finds the correct library files.

B) Does *not* store a `REM`ark line for each module.

C) *Does* have the ability to track a “tree” of subdirectories. To utilize this, you must set a the `TREEDIRS` configuration parameter (p. 300). Configure `TREEDIRS Y`, and optionally configure the `WORKDIR` parameter (p. 304) to tell TLIB which directory is the “root” of your “tree” of working directories.

If tracking is enabled and you extract a file with the E or EB command, you can optionally configure TLIB so that you will get the version of the file indicated by the entry in `TLIBWORK.TRK`, instead of the latest “trunk” version). This behavior is selected by configuring:

```
  PROJLEV =
```

However, by default (with `PROJLEV` not configured), the extract will get the latest trunk version, just as if version tracking were not enabled (which is appropriate for most users).

Of course, you can always extract a different version by specifying a specific version number with the ES command, and the `TLIBWORK.TRK` entry will be adjusted accordingly, regardless of how the `PROJLEV` configuration parameter is set.

Automatic branching

`TLIBWORK.TRK` enables TLIB to “know” what versions of each module you are working on, so that the U (update) command can automatically store them as branch versions when that is appropriate.

When would that be appropriate? When storing the new version as a “trunk” version number would effectively “undo” changes from an earlier version. In other words, if you started with something other than the latest trunk version.

Thus, for example, if the latest trunk version was number 23, and you extracted version 22 and made changes to it, when you store the new version it should be as version 22.1 (a branch from version 22), since the new version does *not* include the changes that were made to create version 23. It would be a mistake to store the new version as 24 (the next trunk version), since that would undo whatever changes were in version 23.

However, if tracking is not enabled, then TLIB will not have a record of what version you are working on, so it will store the new, modified version as version 24 (unless you use the “US” command and manually force it to store the new version as a different version number).

By enabling version tracking (configuring `TRACK Y`), you can avoid this pitfall.

To control TLIB's automatic branching feature, configure the `AUTOBRNCH` parameter; see p. 298.

To disable automatic branching but still have TLIB maintain the `TLIBWORK.TRK` file, you can configure:

```
  PROJLEV *
```

Extracting to a temporary file

Suggestion: it is advisable to configure TLIB to track only certain files (your source files). So, you should add something like the following to the TLIB configuration file (or let TLIBCONF do it for you):

```
  TRACK N
  IF *.C,*.H,MAKEFILE.*
    TRACK Y
  ENDIF
```

This will allow you to extract a temporary copy of an old version of a source file without causing an entry to be added to the tracking file.

For instance, if you wished to examine the first version of a file called `FOO.C`, you could do “`TLIB EBS FOO.C1 1`” (assuming that your `LIBEXT` configuration parameter is chosen such that `FOO.C` and `FOO.C1` both “map to” the same TLIB library file). (See also p. , for another way to extract to a temporary file.)

Since "`*.C1`" files are not tracked, `FOO.C1` will not be recorded in the tracking file.

Note that the `TLIBWORK.TRK` file is always kept as read-only, to prevent its accidental deletion.

Also note that even if you store the various versions of `TLIBWORK.TRK` in a TLIB library, they will not be tracked. That is, TLIB will not put a "`TLIB-WORK.TRK`" entry in the `TLIBWORK.TRK` file, itself.

Note also that you can explicitly use `TLIBWORK.TRK` as if it were a file list or snapshot file, extracting all the modules named in it. Use the "`@`" syntax, like this: "`TLIB ES @TLIBWORK.TRK`".

Semi-custom software

You can you use TLIB's version tracking to help simplify maintenance of multiple, customized versions of your software. This is explained later.

Saving disk space

When you are not working on a particular customized version, you can save disk space by deleting the source code, keeping only the tracking file and the TLIB libraries. See "Saving Disk Space", p. 199.

Merging fixes

The M ("migrate") command can be used to merge changes from one level of code into another, all at once. For instance, if you support "semi-custom" software, with one "standard" level and many customized levels, you can use the M command to migrate fixes from a new "standard" level into any of your customized levels.

The M (migrate) command uses TLMERGE (or DIFF3) where necessary to merge fixes into customized modules, and copies those modules which have not been customized, and records the migration history in specially-formatted TLIB comments so that future migrate commands can tell which changes have already been migrated, to avoid erroneous attempts to merge the same changes again.

This is described more fully under "M - Migrate Changes" (p. 178).

Configuration

The following configuration parameters are provided to support basic version tracking: `TRACK`, `TREEDIRS`, `WORKDIR`, `TRACKEXT`, `DOTDOTOK`.

Additionally, the following configuration parameters are provided to support named project levels: `CREATETF`, `LEVEL`, `PROJLEV`, `REFSUBDIR`.

Tracking File Terminology

"**Version Tracking File**"
(or "Tracking File" for short)

A Tracking File is a simple "flat file" database used by TLIB to track version numbers and other information about the source modules which you maintain under version control. Tracking files are usually named "`TLIBWORK.TRK`" (but the `.TRK` extension can be changed via the `TRACKEXT` configuration parameter).

A tracking file resides in either a "work directory" or a "reference directory" which may (or may not) also contain copies of the tracked source files. By default, the current work directory is usually the current directory (or perhaps its parent directory), but you may wish to use the `WORKDIR` configuration parameter to change it.

If `TREEDIRS Y` is configured, then the "work directory" (and the "reference directory") is just the "top" directory in a tree of related subdirectories, and the tracking file records not only the module names, but also the "relative paths" from the top directory to the subdirectories in which the tracked modules reside.

"**Work Directory Tracking File**"

This is a tracking file which resides in the current work directory and is used to keep track of the current versions of the modules which you have (or last had) copies of in that directory. If `TRACK Y` is configured, then there is always one of these. If the work directory tracking file doesn't exist, then TLIB will create it.

"**Current Project Level Tracking File**"
(or "project level file" for short)
(or "PROJLEV File" for even shorter)

This is a tracking file which resides in the "reference directory" of the current "named project level," if any. If you are using named project levels, then you must use `LEVEL` configuration parameters to tell TLIB about them, and you must also use the `PROJLEV` configuration parameter to select which of your project levels is the current one.

One of the things which the `LEVEL` configuration parameter tells TLIB about each project level is where its reference directory is.

Different project level files may be “current” at different times, but at any one time there is only one current project level file.

Named project levels are used for “advanced version tracking” (explained below).

“**Other Project Level Tracking Files**”

If “promote-chains” (multiple assurance levels) or customized version levels are used, then one or more other (non-current) tracking files may also be accessed by TLIB (this is explained later). However, only the current project level file (and the work directory tracking file) will usually be modified by TLIB. (Exception: when “`s=Old`” is configured for the current level.)

What Are Tracking Files and Named Project Levels Good For?

The short answer to this question is that TLIB uses Tracking Files to implement several important features, one of which is “Named Project Levels.”

Named project levels are principally used for just one thing: managing multiple variants of the same project or program. If there are two or more versions of your project or program which are being actively changed, then you need to use TLIB's named project levels. But if there is only one version (i.e., “the latest version”) that is subject to modification, then you probably do not need to use TLIB's named project levels.

1) For version labeling.

Version labeling is the process of tying together all the versions of all the modules that make up a product.

A version label (or “snapshot”) is simply a file which records the version number of each module that is part of a release of your product. That is, the version label file is a file that contains [module name, version number] pairs.

Creating a version label ensures that (using TLIB) you can easily reconstruct that release of your product at a later date.

Before you release a version of your software, make sure that you have updated TLIB libraries (checked-in all the source modules). Then use TLIB's S (snapshot) command to create a version label file. The S command consults your project level tracking file(s) to determine the correct version numbers.

Alternately, in simple development environments, creating a version label can be simply a matter of making a copy of the tracking file.

Although the snapshot and tracking files have different formats, they contain similar information, and TLIB can use them interchangeably. So, if a tracking file exists which already contains the needed version number information, you can simply copy it instead of using the “S” command. This will work...

o if you're using “basic version tracking” (no named project levels), so that a copy of the work directory tracking file is usable as a version label; or,

o if you have just one named project level, so that a copy of its tracking file is usable as a version label; or,

o if you have several project levels but the current project level is “fully populated” (fully-populated vs. “sparse” project levels are described later), so that a copy of the current level's tracking file is usable as a version label.

To save the copy, you can simply use the DOS copy command to copy the tracking file to another file with a different name. Or you can make it more compact by using COPYTRAK with the “-s” option.

Other users will need to use TLIB's S (snapshot) command to make the version label, rather than simply copying the tracking file. This is required when the needed information is distributed among several tracking files (due to “sparse” chained project levels, as explained later), or when you have tracking disabled for some (or all) of your source files.

Regardless of how you create you version label files (with the S command or by copying the tracking file), you'll probably find it convenient to store them in a TLIB library. With TLIB, you can keep you version labels under version control! This has the advantage of reducing file clutter and allowing you to associate comments with the version label file.

2) For automatic branching.

Simply by setting the correct current project level, you can ensure that TLIB will utilize the correct branch development paths when updating TLIB libraries and extracting source modules. This works either with or without check-in/out locking enabled.

3) For keeping multiple assurance levels of code (“promote chains”).

This is the classic “staged” approach to managing large software development projects. For example, you could have a development level, a test level, and a release level. The AP ("promote") command lets you "promote" the status of a particular version of a module from one level to the next. Think of "promoting" a source file from the development level to the test level as "throwing it over the wall" to the QA team.

4) For change migration.

TLIB 5.50 supports the "M" (migrate changes) command (p. 178), which can use version tracking files to make it easy for you to migrate changes from one level of code to another, even for entire library levels. For instance:

o If you used a branch project level to do maintenance on an early release of your software, you could use M (migrate) to apply all the fixes in all the modules onto the latest release.

o If you used a branch project level to maintain a customized version of your software, you could use M (migrate) to add the latest "standard" modifications to the customized version.

Supported Development Environments for Version Tracking

Several different environments are supported, with one, two or more tracking files used, depending upon the environment.

1. Single programmer, single development line, one current code level.

This is the simplest environment. There is only one work directory (or perhaps a tree of subdirectories) in which source code normally resides.

Check-in/out Locking is normally disabled in this environment (since the purpose of check-in/out locking is to keep track of which programmer is working on which modules, which is unnecessary in this environment).

TLIB maintains just one tracking file (`TLIBWORK.TRK`), which resides in the work directory (or the "top" directory of the tree). (This is "basic version tracking.")

You need not configure `PROJLEV` or `LEVEL` at all.

In fact, even the use of version tracking is optional. You may be perfectly satisfied to disable tracking (`TRACK N`) and simply use the S (snapshot) command to create version labels when you make a new release. This is (roughly) how TLIB 4.12 did things.

2. Single programmer, several development lines (customized versions).

This environment is similar to #1 except that there is one directory (or a tree of subdirectories) for each customized version.

Check-in/out Locking is normally disabled in this environment.

TLIB maintains a tracking file (called `TLIBWORK.TRK`) for each of the customized versions. Each tracking file resides in the corresponding work directory (or the "top" directory of each tree). However, only one tracking file is in use at any given time (the tracking file used is the one in the current work directory, or the top of the tree of directories).

One approach is to configure "`PROJLEV =`" and not use named project levels at all. This is the "basic version tracking" approach.

However, for more flexibility you'll probably prefer to define each work directory as the reference directory for a named project level. This allows you to differentiate between the "standard" level and the "customized levels," so that customized modules are automatically stored as branch versions in the TLIB libraries. You would use the `b=1` option on the `LEVEL` configuration parameter for each customized version to "force" branching when you store modified modules for that version, along with the `c=`*nn* option to select consistent branch numbers for each customized version

Note that with this approach it is not necessary for you to explicitly configure the `PROJLEV` or WORKDIR parameters, since TLIB will notice that you are working in the reference directory for one of your configured project levels, and will automatically set `PROJLEV` and `WORKDIR` accordingly.

3. Two or more programmers working together using a network (LAN) and working on a single line of development.

Check-in/out Locking is normally enabled in this environment, to keep track of which programmer is working on which module(s).

The TLIB libraries reside on a shared-access network file server, and individual programmers check source modules in and out from the server. Each programmer has his own, private work directory (or tree of subdirectories), usually on his local hard disk.

In this environment there are several tracking files maintained by TLIB: one tracking file for each of the programmers' work directories (or trees of subdirectories), and at least one shared "project level" tracking file which resides on the LAN file server. Thus, when any particular programmer is using TLIB Version Control, TLIB must simultaneously utilize two tracking files.

If you need to "stage" development (e.g., between "development," "test," and "release" levels), then there will be several named project levels, with programmers first storing their changed modules at the rough level (e.g., “development”), and later “promoting” them to a higher assurance level. The linkage between levels is established via the `p=` and `i=` options on the `LEVEL` configuration parameters.

The “current project level tracking file” resides on the Network file server and tracks the current level for the whole project.

The “work directory tracking file” resides in the programmer's private work directory (or in the top directory of the tree of working subdirectories).

All of the tracking files are named `TLIBWORK.TRK`.

The location of the (shared-access) project level tracking file is determined by the PROJLEV and LEVEL configuration parameters.

4. Two or more programmers working together using a network (LAN) and working on multiple lines of development.

This environment is similar to #3, except that there is one directory (or a tree of subdirectories) for each customized version.

Check-in/out locking is enabled.

There is (at least) one project level, and, hence, one project level reference directory and tracking file on the server for each of the customized versions of the software. However, there is still just one directory (or tree of subdirectories) on the server for all the TLIB libraries and lock files.

For each of the project levels, you'll have to configure a `LEVEL` parameter in your TLIB configuration file, and for each customization level you should specify the “`b=1`” and “`b=`*nn*” options on the `LEVEL` parameter (you would not do this on the `LEVEL` parameter(s) for the standard version of the software).

This is a rather complex development environment, and it is discussed more thoroughly later on.

Where the Files and Directories Belong

With TLIB, you will have only one set of TLIB libraries, regardless of how many project levels (variants of your software) you have. In a multiple-programmer project, the TLIB libraries will reside in a shared directory on the file server.

In addition, if you use named project levels (to manage multiple variants of your software), then there will also be one project level reference directory for *each* of the named project levels. In a multiple-programmer project, this, too, will be in a shared directory on the file server.

Note that since the TLIB libraries are not tied to any specific project level, they should not normally reside in any of the project level reference directories, nor in a subdirectory of any of the project level reference directories.

Handling Semi-Custom Software With Basic Version Tracking

Note: this chapter is soon to be revised; you may call for free technical support if you need to use TLIB to manage customized software.

How can you use TLIB's tracking file to simplify maintenance of multiple, customized versions of your software product?

If yours is a large project, in which multiple programmers will be working on (different parts of) the same customized version of the project, you will need to utilize the “advanced version tracking” and named project levels, which are explained later.

If yours is a small project, and you are the only programmer, you may be able to use the basic tracking support, without named project levels, as described in this section. However, you still may prefer to use advanced version tracking (with named project levels), for greater flexibility.

The constraint for “basic tracking” is this: you must have *one* subdirectory (or one tree of subdirectories) for your “standard” version, and *one* for each of your customized versions. That one directory serves as your work directory when you are working on the associated customized version of your software project.

You needn't actually have copies of the source files in each of these directories at all times, but you must have the directories, because each of the customized versions will have a `TLIBWORK.TRK` file in its work directory which records the modules and version numbers for that variant of your software product.

When you are working on a particular customized version of your software product, you must make that customization's work directory the “current” directory. It is the directory which contains the tracking file (`TLIBWORK.TRK`) for that customized version of your software product. Then when you use TLIB to extract or update your files, TLIB can use the current `TLIBWORK.TRK` file to retrieve the right versions - and store new modifications in the correct branches.

To make TLIB retrieve the versions recorded in `TLIBWORK.TRK` while extracting, you'll need to configure:

```
 PROJLEV =
```

To create the initial tracking file for the “standard” version of your software, you can simply configure TLIB to track your source files, then create the work directory, and then extract all the correct versions (use the E or EB command, perhaps with the `@`*snapshot* or `@`*filelist* syntax). Or, if you already have the source files in the directory, you can use the U command to update the TLIB libraries; those modules which haven't changed will not actually be updated (unless you configure `FORCEU Y`), but the entries in `TLIBWORK.TRK` will still be added.

To create the initial tracking file for a customized version, you must first create a work directory for the customized variant, then you must create its tracking file using COPYTRAK's “`-c`” option (note that this is *not* necessary if you are using advanced version tracking and named project levels).

“`COPYTRAK -c`” simply copies a tracking file and adds `c=N` fields for each module to the copy. This field tells TLIB to force creation of a branch if you make a new modification (customization) while working on this customized version of the module. (With advanced version tracking, the `b=1` option on the `LEVEL` parameter is used to force branch creation, instead.)

Tracking file format

A `TLIBWORK.TRK` “tracking file” is a fixed-line-length ASCII text file which TLIB uses as a simple database.

It consists of an indefinite number of lines (records), each exactly 128 bytes in length (including carriage-return and line-feed). The 126th byte of each line is always a period. The first 125 bytes are available to store information. You can think of the tracking file as a simple “flat file” database, with fixed record size and variable-length fields within each record.

This information about the format of version tracking files is provided to satisfy your curiosity.

It should not usually be necessary for users of TLIB to examine or modify a TLIB tracking file, but since it is a normal, ASCII text file, you can do so if the need arises.

Each record (line) contains a “key” and one or more “fields.”

The “key” is the first thing in the record. It begins in column 1 and ends with a blank.

The fields are of the form “*a*=*value*” where “*a*” is a single lower-case letter, and “*value*” is text which does not contain any blanks. The fields are separated from one another by a single blank, and the unused portion of the record is blank-filled.

The first record (line) in a tracking file is special. It begins with a key of “`!!*`”, and it is used to record a “project level name.”

Except for the first record in the file (which has the special “`!!*`” key), the key is always a file name (possibly prefixed by a “relative path”) for a file which you keep under TLIB version control. Under DOS and OS/2, the key is case-insensitive (that is, it does not matter whether it is upper-case or lower-case), because the file names of these operating systems are case-insensitive.

Summary: Tracking files contain two kinds of records:

1) The first record ("header" record) in a tracking file is special. It has the special key "`!!*`", and currently has only has one field:

```
  n=project-level-name
```

This field is used only in the special "`!!*`" record at the beginning of each tracking file, and are it is normally only present in project level tracking files (rather than work directory tracking files). It is used as a consistency check by TLIB, to ensure that the `d=` field in the `LEVEL` configuration parameter for the project level indicates the correct directory.

The name should match the `n=` field of the `LEVEL` configuration parameter for the project level.

For example, if file `F:\TRUNKS\TLIBWORK.TRK` contains `n=TRUNK` in the `!!*` record, then the TLIB configuration file should contain the line,

```
  LEVEL n=TRUNK d=F:\TRUNKS\ ...
```

2) Most of the records (one per tracked source file) have the source file name as the key, and contain one or more of the following fields:

```
  v=    version number (always present, except for "deleted"
names)
  c=    customization flag (occasionally present)
```

It may also contain these "internal use only" fields with which you needn't concern yourself:

`l=`	*library size when module was extracted*
`t=`	*"tipness"*
`s=`	*lock status (unimplemented)*

Each record is used to store information about a particular module being tracked. The key is a file name (possibly prefixed with a "relative" path) for a file which you keep under TLIB version control.

Under DOS & OS/2, the key is case-insensitive.

Details:

`v=`

The “`v=`” field is usually present. It contains the version number for this module, in standard TLIB 5.50 syntax. Note that it is always a “fixed” (“non-floating”) version number (that is, it cannot contain an asterisk).

If the `v=` field is absent, then the record is just a place holder for the key, in case you should someday add that file name to the tracking file (perhaps with the “A” command). These place holder records are created when the “AD” command is used to delete file names from a project level. The use of place holder records enables TLIB to ensure that records will not “move” in the tracking file, a constraint which lets TLIB create and cache its internal indices more efficiently.

Note: The rest of this chaper is just included for completeness; you probably will never need this information.

`c=`

The “`c=`” (“customization”) field is not often used, since the `b=1` option on the `LEVEL` configuration parameter is usually more convenient. If `c=` is absent, the TLIB U command will attempt to extend the library with the next sequential version, only creating a new branch if necessary. If `c=N` then this module has not yet been customized, but if you update the TLIB library with a new version then a new branch will be created even if the next sequential version does not yet exist (this is useful as an initial value for the records in the tracking file for a “newly-cloned” variant of your software product). If `c=Y` then TLIB has already created a new branch for this variant of the module; it should never be necessary to do so again.

`t=` *and* `l=`

These fields are used by TLIB to optimize the operation of the F (fast) suffix option on the E (extract) command. These fields are not necessary for the operation of TLIB, and you will not “break” anything by deleting these fields, except that if these fields are deleted then the “fast-extract”/“freshen” commands (EF, EBF, EFS, etc.) may do superfluous extracts.

The “`l=`” (library-file-length) field is used to record the length of the TLIB library file at the time that the corresponding source file was last extracted into that directory. When TLIB stores a new version number for a module in a tracking file, it either deletes the `l=` field (if the source file is *not* up-to-date in that directory), or it stores the length of the library file in the `l=` field (if the source file *is* up-to-date).

TLIB libraries always grow and never shrink, (because TLIB only appends to them, without changing the old portion), so if someone has stored a new version of the source file, the `l=` fields in each of the tracking files will indicate the wrong (shorter) lengths, which implies that the source file might need to be refreshed.

More precisely, if you are specifically extracting the version recorded in the tracking file (e.g., via the “`TLIB EBF @TLIBWORK.TRK`” syntax), then the source file needs to be refreshed only if the `l=` field is missing; but if you are extracting some other version then the source file needs to be updated if the library has grown since the current version was extracted.

The “`t=`” (tip-version-indicator) field is used in combination with the `l=` and `v=` (version-number) fields to determine when TLIB can safely skip a file while doing “fast-extracts” (EBF, EBFS, etc.), even if you do fast-mode extracts with different specified “floating version numbers” (containing asterisks) on different occasions.

The details of how TLIB creates and uses the `t=` fields are rather complex. The presence of the `t=` field just makes it possible for TLIB to determine under a greater variety of circumstances when it can avoid unnecessary extracts during “fast extract”commands.

There are three possibilities for the `t=` field:

1) If the `t=` field is absent, it means that nothing is known about the “tipness” of the version, or it is not a tip version.

2) If `t=L`, it means that the specified (“`v=`”) version number is the very last version in the library file (so long as the library file is still equal to the length indicated by the `l=` field).

3) If `t=T`, it means that (if the library file is still the length indicated by the `l=` field) then the specified version number is a tip version, though not the last version in the library.

This will be true, for example, if the `v=` version is the latest trunk version, but there is a branch version that is more recent. It will also be true if the `v=` version is the latest version in a particular branch, but another trunk or branch version is more recent.

The absence of the `t=` field is never a serious problem. At worst, it will cause TLIB to do unnecessary re-extracts when a fast-extract command is done. So, if you build a tool which directly manipulates TLIB tracking files, it can just delete the `t=` and `l=` fields and not worry about it.

Advanced Version Tracking & Named Project Levels

Note: this chapter is soon to be revised; you may call for free technical support if you need to use TLIB to manage customized software.

“Advanced Version Tracking” is the use of one or more “named project levels” to represent the various lines of development or staging levels for your project. This feature provides automatic branching and automatic version labeling for multiple-programmer teams in networked environments, even when there are multiple lines of development or several different “current” levels of code.

To support named project levels, TLIB 5.50 simultaneously uses at least two tracking files: the work directory tracking file, and the project level tracking file.

Work Directory Tracking File:

One tracking file is used to track what is in your “work” directory. It is usually named `TLIBWORK.TRK` and it resides in the work directory (or in the top directory of a “tree” of work directories). In a networked environment, each programmer will have one of these in his private work directory. This tracking file is sometimes called “the work directory tracking file.”

The location of the work directory tracking file is determined by the `WORKDIR` configuration parameter (which can reference an environment or autoset variable, if you wish).

`WORKDIR` usually defaults to “`.\`” (the current directory), but under some circumstances it defaults to a parent directory of the current directory if you've configured `TREEDIRS Y`.

Note that the default may not be what you want if you specify explicit paths for your source files when using TLIB!

The only reason TLIB needs a work directory tracking file is to keep track of what versions of your source files you currently have. This is necessary for automatic branching, and to implement the F (fast/freshen) suffix option for the E (extract) command.

Project Level Tracking File:

The other tracking file is used (perhaps simultaneously) by all of the programmers on the team. It is also usually named `TLIBWORK.TRK`, but it resides in a shared-access "reference" directory on the network, and it tracks which is the current version of each module in a "current project level."

This tracking file is called a "project level tracking file."

Note: If you are a single programmer working alone, and you only have one current version of your programs, you may not need to use named project levels and project level tracking files.

The location of the project level tracking file is determined by the `PROJLEV` configuration parameter, which is normally set to a "set" name, which is usually resolved by a `SET` configuration parameter or a `SET` command in your autoset file (but can also be resolved by an environment variable). There is an example below showing how to set this up.

There can be many project levels (and thus many project level tracking files), but only one of them is the "current" project level tracking file, as determined by the `PROJLEV` parameter.

When you Extract (check-out) a module, the version number for that module in the project level tracking file determines which version is retrieved (unless you use the S suffix with the E command, to specify a particular version). When you Update a library (check-in a source file), the version number from the work directory tracking file tells TLIB what version you started with when you began editing that file.

This, in turn, determines the new version number for the module (which may be a branch). However, a warning will be issued if the version numbers in the two tracking files are not consistent (this is rare, but it could happen, for instance, if someone else had disabled locking and stored a new version without telling you).

Setting Up Your Project Level Tracking Files

Note: this chapter is soon to be revised; you may call for free technical support if you need to use TLIB to manage customized software.

Setting up Work Directory Tracking Files:

There is no set-up required for work directory tracking files. They will be created and updated as necessary when you extract source files and update TLIB libraries.

Setting up a Project Level Tracking File for the "mainline" version:

In a multi-programmer environment, you'll need to set up least one project level tracking file and its associated reference directory (for tracking "trunk" or "main-line" versions). This requires the following steps:

A) Create a "reference directory" on the file server, named mnemonically. If your file server is the "`F:`" drive, you might name this directory `F:\TRUNKS`. If you use a "tree" of subdirectories to contain your source files, create the needed subdirectories, too. TLIB will not create the directories automatically (unless you configure `MAKEDIRS Y`, see p. 315).

For example, suppose you have a project for which source files are kept in a main directory and also in two subdirectories called `IOSTUFF` and `SCREENS`, respectively. Then you could create the reference directory and its subdirectories like this:

```
MD F:\TRUNKS
MD F:\TRUNKS\IOSTUFF
MD F:\TRUNKS\SCREENS
```

Note: If you use a "tree" of subdirectories, configure `TREEDIRS Y`.

B) Invent a mnemonic name for this project level. You may want to use the name of the directory from step (A). The name should be entirely alphanumeric (no punctuation characters), and all upper-case.

C) Edit your TLIB configuration file (usually `TLIB.CFG`).

i) Add a `LEVEL` configuration parameter to your TLIB configuration file, like this. At a minimum, this parameter must specify the name (`n=`) and directory (`d=`) for the project level. You may also wish to configure `a=Y` to cause modules to be automatically added to the project level tracking file.

For example, if the name that you invented was `TRUNKS`, and the directory you created in step (A) was `F:\TRUNKS\`, then an appropriate `LEVEL` configuration parameter would be:

```
    LEVEL n=TRUNKS d=F:\TRUNKS\
```

The directory name must be a fully-qualified drive and path ending in a back-slash. (E.g., "`F:\TRUNKS\`", not `TRUNKS\` or `F:TRUNKS\`).

Note: see "Configuring Your Project Levels" (p. 141) for more information on configuring the `LEVEL` parameters.

ii) If you are using a tree of subdirectories for your source files, you should also add "`TREEDIRS Y`" to your configuration file:

```
    TREEDIRS Y
```

iii) You should also configure the `PROJLEV` parameter.

If you will have only one project level, set `PROJLEV` to the name you chose in step (B):

```
    PROJLEV TRUNKS
```

If you will have more than one project level, it is better to set `PROJLEV` to reference a "set" name. Then you can select the current project level by defining the name in your `AUTOSET.BAT` files with a `SET` command (or with an environment variable).

You use what whatever name you like. For this example, we chose "PROJECT":

```
    PROJLEV %!!PROJECT%
```

Note that the "`!!`" causes TLIB to display an error message and halt if the name `PROJECT` is not defined.

iv) Add the following temporary configuration parameter to the end of the TLIB configuration file:

```
CREATETF Y
```

The name “`CREATETF`” is short for “Create Tracking File.” This parameter will cause TLIB to automatically create the project level tracking file if it does not already exist.

When you are finished setting up your project levels, it is a good idea to delete this parameter or change it back to the default, `CREATETF N`.

v) Save the modified TLIB configuration file.

D) Or, If you did not configure `CREATETF Y`, then you must manually create a `TLIBWORK.TRK` tracking file in the new reference directory.

The easiest way to do this is to use the POKETRAK utility, setting the `n=` field in to first (`!!*`) record to the name you chose in step (B).

For example:

```
POKETRAK F:\TRUNKS\TLIBWORK.TRK !!* n=TRUNKS
```

This creates a 1-record tracking file, `F:\TRUNKS\TLIBWORK.TRK`, with your chosen project level name recorded in it.

E) If you configured `PROJLEV` to a “set” name reference, like `%!!PROJECT%`, then use an environment variable or (more likely) an autoset file “set” command to define the name which you chose in step (C,iii) to be the project level name you chose in step (B).

For our example:

```
SET PROJECT=TRUNKS
```

Now, when you create or update your TLIB libraries, the project level tracking file (`F:\TRUNKS\TLIBWORK.TRK`) will be used by TLIB to record the current version numbers of each source module.

Setting up Project Level Tracking Files for branch versions:

If you will need to make changes to more than one version of your software, then you need an additional project level tracking file and reference directory for each such variant.

There are many reasons for having additional project levels of your software. For instance,

o Customized versions for special customers

o Multiple assurance levels, such as Release vs. Test vs. Development

o Multiple supported releases of the product

o Maintenance or "bug fix" levels

To create a branch project level, you follow much the same procedure as you followed to create the mainline project level. The differences are in steps (C) and (E), and perhaps (D):

C) In your TLIB configuration file:

You will now have a `LEVEL` parameter for each of your project levels, defining each of the project level names and their corresponding reference directories.

In the `LEVEL` parameter for each branch/customization level:

o Specify the `i=`*names* field of the `LEVEL` configuration parameter to indicate which "main-line" level(s) the new level is based upon.

o Specify the `b=1` and `c=`*nn* fields of the `LEVEL` configuration parameter, to force branching in the level if it will be used for alternate development paths (bug fix levels, customization levels, etc.). We recommend that you select values of *nn* which are even multiples of 5 or 10. However, you should not specify `b=1` if the level is for "staged" development, such as a "development" or "test" level for mainline code, since revisions at that level should be stored as trunk versions.

For step (C,iii), above, you should configure `PROJLEV` to be a current project variable, so that you can change the current project via an environment variable or autoset command.

(There is no need to have more than one `TREEDIRS` configuration parameter.)

E) Whenever you plan to use TLIB to extract (check-out) a source module or update a TLIB library (check-in a module), you must set the current project level to be one of the project levels you have configured. If you fail to do so, TLIB will exit with an error message (due to the “`!!`” which you configured in the `PROJLEV` parameter in step (C)).

D1) You can configure “`CREATETF Y`” to make TLIB automatically create the new project level tracking files. Alternately, you can create an empty (0 to 3 byte) tracking file with a text editor, or with a DOS command like “`echo.>\refdir\tlibwork.trk`”. TLIB will replace the empty tracking file with a proper one. Or, you can use the POKETRAK utility to create it, like this:

```
  POKETRAK F:\CUSTOM1\TEMP.TRK !!* n=CUSTOM1
```

D2) As an alternative to using the `b=1` option, you can create your new project level tracking file with the COPYTRAK utility, using the `-c` switch (run COPYTRAK with no parameters for help).

Start with the tracking file for the predecessor release, from which this new variant is derived. For example:

```
  COPYTRAK -c F:\TRUNKS\TLIBWORK.TRK F:\CUSTOM1\TLIBWORK.TRK
```

Then set the name in the special “`!!*`” record with POKETRAK. For example:

```
  POKETRAK F:\CUSTOM1\TLIBWORK.TRK !!* n=CUSTOM1
```

Creating the new tracking file this way (with COPYTRAK) makes the new level initially “fully populated” (as opposed to “sparse”), which may be convenient for some users. Alternately, you can populate the a new branch level with the AF command. (Sparse vs. fully populated levels, and the AF command, are explained later.)

Setting up for branch versions with “chained” Project Levels:

Unless you used COPYTRAK to create your branch/customization project levels, then the initial tracking files for those levels are empty. All the source files are simply “inherited” from the parent level. As you customize various source files and store the new (branch) versions into the TLIB libraries, the branch/customization project level tracking file will grow to record the customized source files.

This is called a “sparse” project level, because it only explicitly lists the modules which differ from the parent (standard) level.

Alternately, if you use COPYTRAK (as described in D2, above) to set up your branch/customization level tracking files, the initial tracking file lists all of your source files - both the customized ones and the non-customized ones. Thus, if a new version of a module is stored in the “standard” project level, it will not affect the customized level.

This is called a “fully-populated” project level.

Each approach has both advantages and disadvantages. The main advantage of fully-populated levels is that changes to the standard version will not interfere with developers who are working on the customized version. The disadvantage is that someone will eventually have to migrate those changes into the customized variant. However, TLIB's M (migrate) command usually makes this a fairly painless process.

However, to avoid the migration chore, at least some of the time, you may prefer to use sparse project levels. A sparse “customized” project level contains only those modules which have actually been customized, with the rest of the modules “inherited” from the “predecessor” (a.k.a. “standard” or “parent” or “base-line”) project level.

For a sparse project level to work correctly, you must tell TLIB which other level is the parent level upon which the sparse customization level is based. You do this via the `i=` field in the `LEVEL` configuration parameter for the customized level. Simply set the `i=` field to the name of the standard (base-line) project level. Then if your current project level is the customized level, when you extract a module its version number will be determined by the customized project level tracking file if the module is listed there, and otherwise by its entry in the standard project level.

In other words, the two project levels are “chained” together by the existence of the `i=` field.

To set up for chained project levels, the only difference is in step D, above. Instead of creating an initial customized project level tracking file which contains all the modules (D2), you create an empty tracking file (D1). If you have configured `CREATETF Y`, then TLIB will do it for you; otherwise, you can create it manually with the POKETRAK utility, setting the n= field of the “!!*” (header) record to the name of the customized level:

D1) For example:

```
  POKETRAK F:\CUSTOM1\TEMP.TRK !!* n=CUSTOM1
```

One thing to be aware of if you use this approach: a version label is no longer just a copy of the tracking file. Instead, it must also contain the modules “inherited” from the base-line project level.

For this reason, TLIB includes the S (“snapshot”) command. The S command can create a version label file that lists both the customized source files (found in the current level) and the uncustomized source files (found in the parent level). That is, it also lists all the “inherited” modules from the base-line project level (and/or promote levels).

Sorting a tracking file

COPYTRAK can also be used to sort tracking file, if you specify the -s option:

```
   ATTRIB -R F:\CUSTOM1\TLIBWORK.TRK
   REN F:\CUSTOM1\TLIBWORK.TRK TEMP.*
   COPYTRAK -S F:\CUSTOM1\TEMP.TRK F:\CUSTOM1\TLIBWORK.TRK
   DEL F:\CUSTOM1\TEMP.TRK
```

TLIB doesn't care whether the tracking file is sorted or not, but you may wish to sort it for aesthetic reasons. Do not do this, however, when another user is running TLIB, since TLIB “remembers” the locations of the various records in a tracking file, and does not expect them to move.

How TLIB Uses the Tracking Files

Which tracking files are updated by TLIB, and when

When a module is retrieved, the version number is stored in the work directory version tracking file. The version number is not, at that time, stored in any other tracking files.

When the TLIB library for a module is updated (a new version is stored), then both the work directory tracking file and the appropriate project level tracking file (if any) are updated with new version numbers for the module.

The "appropriate" level is usually the current level, but if the module was retrieved from an "inherits-from" level, then that level may be used, instead, depending upon how you have set the optional `s=` field on the `LEVEL` configuration parameter for the current level.

How the Various Version Tracking Files Are Used By TLIB to Determine the Version Number of a Retrieved Module

If TLIB is retrieving a module and `PROJLEV` is set to a current level which is configured with `p=` (promotes-to) and/or `i=` (inherits-from) field, then TLIB may also utilize these "predecessor" levels to determine the version number of the extracted source module.

If the user explicitly specified the desired version, or if the module is listed in the current project level file, then the `p=` and `i=` fields have no effect. Otherwise, the `p=` and `i=` levels are examined, looking for a higher assurance level version of the module. If the module is not found in any of the levels, then the latest trunk version is used.

Step #0: If the user specified a particular version (either directly or through the use of a file list), then that version number is used and the version tracking file(s) are ignored.

Step #1: If "`PROJLEV *`" is configured, or if `PROJLEV` is not configured at all, then the latest trunk version of the module is used.

Step #2: If "`PROJLEV =`" is configured, then the version specified in the current work directory tracking file is used. If the module is not listed in the work directory tracking file, then the latest trunk version of the module is used.

Otherwise, `PROJLEV` is the name of the current project level, so...

Step #3: If the module is listed in the current project level tracking file, then that entry is used to determine the version number of the retrieved module.

Step #4: Otherwise, if there is a `p=` field, then that level is examined to determine the version number of the retrieved module.

Step #5: If the module name is not found in the `p=` ("promote-to") level, then the `i=` ("inherits-from") level(s) are examined in the order specified, until a level is found which contains the module.

Step #6: If the module name has not been found in any of the above steps, then use the latest trunk version of the module.

Note that TLIB does not automatically chain-together multiple "predecessor" levels. That is, the only levels that TLIB examines are the current level and the levels specified in the `p=` and/or `i=` fields of the current level. The `p=` and `i=` fields of the predecessor levels are ignored.

[Limit: While trying to determine the version number of a module, TLIB will examine at most 30 "predecessor" levels of the current project level. This seems to be wildly excessive, which is fortunate, since reading lots of project level tracking files would slow TLIB appreciably.]

Configuration Parameters for Version Tracking

The following new configuration parameters are the most important ones provided to support version tracking:

`TRACK` *<Y/N/Maybe>*

Enable/disable basic version tracking. You should use `IF`/`ENDIF` blocks to enable tracking for only those source files which are part of your product. That way, you can check-out "temporary" copies of old versions of source files (naming them "`file.c1`" for instance, instead of "`file.c`") without having them tracked. If you use TLIBCONF to set up your TLIB configuration file, it will create the needed `IF`/`ENDIF` and `TRACK` parameters. Default is `TRACK N`. See p. 297.

`LEVEL n=`*name* `d=`*path* `p=`*name* `i=`*names* `s=`*{Old/New/Q/Changed}*
`a=`*{Y/N/Q}* `r=`*{Y/N}* `b=`*n* `f=`*{Y/N}*

`LEVEL` is used to tell TLIB various things about your named project levels, which are only used for "advanced version tracking." `LEVEL` is explained under "Configuring Your Project Levels" (p. 141).

Note: though the `LEVEL` parameter is shown here on two lines, it must be all on one line (of at most 254 characters) in your TLIB configuration file.

`WORKDIR` *path*

Allows you to specify the "root" of a "tree" of subdirectories which contain your source code. This is the directory which will contain the tracking file, `TLIBWORK.TRK`. You need not specify a drive letter. Since the "tree" of subdirectories cannot span multiple disk drives, TLIB does not track the drive letter. You would not normally configure this unless you also configure `TREEDIRS Y`. Default is usually "`WORKDIR .\`". See p. 304.

`TREEDIRS` *<Y/N>*

Enable/disable tracking of “relative subdirectories.” Enable this if you wish for TLIB to track a “tree” of related subdirectories as one logical unit. There will be only one version tracking file for the entire “tree” of directories, and each “key” will contain a relative path along with the file name and extension for source files which reside in the “lower” subdirectories. If you configure this, you should probably also configure `WORKDIR` (previous page). Default is `TREEDIRS N`. See p. 300.

`PROJLEV` *name*

Used to select the name of a “current project level.” The name should be defined by a `LEVEL` configuration parameter.

Alternately, if name is if the form `%!!`*name*`%`, then you can use an environment variable or autoset “set” command to select the current project level name.

See also p. 299 , and below.

`PROJLEV`

If you want TLIB to always retrieve the latest trunk version of every module (unless you specify an explicit version number), but you do not want to disable TLIB's automatic branching support (which is triggered when you store a modification of an old version for a tracked module), then you can configure `PROJLEV` with no name (or don't configure `PROJLEV` at all, since this is the default).

This would normally be done only for small projects which rarely involve branch versions.

See also p. 299.

`PROJLEV *`

If you want TLIB to always retrieve the latest trunk version of every module (unless you specify an explicit version number), and if you want it to store new versions as trunk version numbers regardless of whether they're based upon the latest trunk version (that is, if you want automatic branching disabled), then you can configure “`PROJLEV *`”.

This would normally be done only for small projects which do not involve branch versions.

See also p. 299.

`PROJLEV =`

If you want TLIB to consult the work directory tracking file to determine the versions to be extracted, then you can configure “`PROJLEV =`”.

In a multiple-programmer environment, this causes TLIB's E (extract) command to always retrieve the version that *you* were last working on, even if someone else has stored a newer version of the module.

This is inappropriate for most users.

See also p. 299.

`CREATETF` *<Y/N>*

The `CREATETF` parameter is used to tell TLIB to automatically create missing project-level tracking files (“createtf” is short for “create tracking file”). You can configure `CREATETF Y` if you would like TLIB to create project level tracking files automatically. However, you must still create the required reference directories manually. See p. 298.

`SET` *name=unquoted-string*

The `SET` parameter defines names to be used by TLIB like environment variables or autoset names. You can reference “set” names of any kind in the TLIB configuration file via `%`*name*`%` or `%!`*name*`%` or `%!!`*name*`%` syntax (use the “`!`” if you want an error message to be displayed if *name* is undefined, or use the “`!!`” if you want a fatal error to occur if *name* is undefined). The three kinds of “set” names (environment variables, autoset file “set” commands, and configuration file “set” parameters) are discussed starting on p. 80. Also see p. 270.

How named project levels can be linked to one another

This section explains how TLIB 5.50 “links together” several different project levels (and their associated tracking files) in an “inheritance chain.”

You'll recall that previous sections explained:

o What a “tracking file” is, and how TLIB uses tracking files to keep track of the version numbers for the various related modules which make up a “level of code” for a project.

o How tracking files fall into two classifications:

1) The “local” tracking file for the current work directory (the “work directory tracking file”), which keeps track of the modules that you have checked-out, and

2) Shared-access “project level tracking files” which keep track of the current checked-in modules at one or more “project levels.” (These project level tracking files are what this chapter discusses.)

o That every project level has an alphanumeric name and an associated “reference directory,” and that for each project level you must provide a `LEVEL` configuration parameter which gives, at a minimum, the name of the project level and the path to its reference directory.

For example, if you had a project level named “test,” and its reference directory was `f:\test\`, then you would configure:

```
LEVEL n=TEST d=F:\TEST\
```

o That the `PROJLEV` configuration parameter is used to tell TLIB which of your project levels is the “current” one. (If you have several project levels, you can use a `%`*name*`%` substitution and a `SET` name defined in an autoset file or environment variable to select between them.)

In this section, we tell how you can describe relationships between the various project levels, such as:

o One (or more) project levels "customize" another project level.

o A project level is a newer version based upon an earlier, stable release.

o A project level is a "lower assurance level," from which modules can be "promoted" to the next "higher assurance level."

Three configuration parameters are used to tell TLIB about the relationships of the various levels of code in your system: `TRACK`, `PROJLEV` and `LEVEL`.

The `TRACK` configuration parameter enables or disables tracking altogether. It should be specified within an `IF`/`ENDIF` block to enable or disable tracking for particular file names.

The `PROJLEV` configuration parameter specifies the name of the "current" project level, which should be one of the project levels described via `LEVEL` configuration parameters.

The `LEVEL` configuration parameters each describe one project level, one of which should be the one named in the `PROJLEV` parameter.

Detailed descriptions of these three parameters are:

`TRACK` *<Y/N/Maybe>*

Configure `TRACK Y` to enable tracking. `TRACK N` (disabled) is the default. `TRACK Y` enables tracking. `TRACK Maybe` (or `TRACK M`) enables tracking only for those files that are already being tracked. See p. 297.

Example:

```
    ! Enable tracking for all C++ source files, except for
    ! those with names starting with "temp".
    TRACK N
    IF *.c,*.h,*.cpp,*.hpp,makefile.*
      TRACK Y
      IF temp*.*
        TRACK N
      ENDIF
    ENDIF
```

`PROJLEV` *name*

where *name* is the name of the current project level, which should be one of the project level names defined via the `n=`*name* field in a `LEVEL` configuration parameter (see below).

There is only one current project level at any given time, so there should be only one `PROJLEV` configuration parameter.

See pp. 299 and 139.

`LEVEL n=`*name* `d=`*path* `p=`*name2* `i=`*names* `s={Old/New/Q/Changed}`
`a={Y/N/Q} r={Y/N} b=`*n* `c=`*nn* `f={Y/N} w=`*{Y/N}*

Note: though the `LEVEL` parameter is shown here on two lines, it must be all on one line (of at most 254 characters) in your TLIB configuration file.

There must be one `LEVEL` configuration parameter for each project level. The required fields are:

`n=`*name* gives the name of the project level; it should match the name in the header of the tracking file.

`d=`*path* gives the path of the reference directory for this project level; the reference directory must contain the project level's tracking file. You should specify a fully-rooted path, including both a drive letter and a leading backslash.

For example, this would be fine:

```
        level n=dev d=f:\devdir\
```

But these are not good:

	<u>what's wrong</u>
`level n=test d=z:`	(no leading backslash)
`level n=rel1 d=g:rel1\`	(no leading backslash)
`level n=cust d=\cusx\`	(no drive letter)

The optional fields are:

`p=`*name2* (Promote-to field) gives the name of the “promote-to level” to which modules can be promoted (when you have verified that they are correct).

Implementation note: for the `p=` level, to implement the AP (promote) command, TLIB has to load into memory all tracking file entries (in contrast to the `i=` levels, where just those that are not also in the current level will have to be loaded). Thus, `i=` may end up being somewhat less “costly” than `p=` linkage.

`i=`*names* (Inherit-from field) gives the name of “inherits-from levels” to which TLIB will refer to find modules that are not defined in this project level. If more than one level name is specified, then the names must be separated by commas (with no spaces).

If the `i=` field lists more than one inherit-from level, then the levels will be searched by TLIB in the order they are specified.

If both `p=` and `i=` fields are specified, then TLIB looks first at the `p=` level, and only examines the `i=` levels if the module isn't defined at the `p=` level. That is, the `p=` level (if any) is also implicitly listed as the first `i=` level. Thus, the following two `LEVEL` configurations are exactly equivalent:

```
        LEVEL n=dev d=d:\dev\ p=test i=rel1,rel0
        LEVEL n=dev d=d:\dev\ p=test i=test,rel1,rel0
```

`r=N` (Automatic reference directory refresh disabled) This means that TLIB will not automatically maintain up-to-date copies of your source files in the reference directory for this level. (You can still use the EF or EBF (fast extract) command to populate the reference directory whenever you wish.) `r=N` is the default.

`r=Y` (Automatic reference directory refresh enabled) The `r=Y` field causes TLIB to automatically store up-to-date copies of your source modules in the reference directory for this level. Whenever you do a TLIB command which changes the project level tracking file entry for a source module, (esp., when you update a TLIB library with a new version for this level), TLIB will also refresh the associated reference copy of the source file in the reference directory.

Note: see also the `REFSUBDIR` configuration parameter.

`f=Y` (Full) indicates that this level is intended to contain a full set of source modules. Promoting (with the AP command) from this level will *copy* the source module into the promote level, but the module will still be listed in this level.

`f=N` (Sparse) indicates that this level is intended to contain only those source modules which differ from the ones in its parent levels. Promoting (with the "AP" command) from this level will *move* the source module into the promote level, deleting it from this level.

The default is "`f=N`" (sparse).

Note that the `f=` field has no effect upon the "top" project level (the level with no `i=` or `p=` fields).

`a=` & `s=` These two similar fields which tell TLIB what you would like done when you store a new version of a source file with the U or N command, in two different situations:

1) when the source file was previously unlisted, altogether; and

2) when the source file was previously only listed in an `i=` level.

`a=`... (Add-new field) tells what you would like done when a new source file (one which was not previously listed in either the current project level or any predecessor level) is stored with the U or N command.

There are three choices: `Yes`, `No`, and `Query`; see below.

`s=`... (Store-to field) tell what you would like done when a new version of an old source file is stored with the U (update) command, but the source file was previously listed only in an "inherits-from" (i=) level, not in the current level. The new version can be listed in either the old (i=) level, or the new (current) level.

There are four choices: `New`, `Old`, `Query`, and query-if-`Changed`; see below.

Warning: "`LOCKING B`" (branch locking) should only be used with project levels for which `s=new` has been configured in the `LEVEL` configuration parameter, or which have a `p=` (promote) linkage, or which have no `i=` links configured, since only the current project level will be locked.

`a=y` (Add-new field = Yes) indicates that if a module is checked-in via the U (update) or N (new-library) command, and the module was neither part of the current project level nor inherited from a parent level (i.e., it was an "untracked module") then the module will be added to the current project level.

`a=n` (Add-new field = No) indicates that if an untracked module is checked-in via the U (update) or N (new-library) command, then the module will *not* be added to any project level. Thus, this module will remain untracked (except in the local tracking file).

If you configure `a=n`, you can still use the A command (add/alter projlev) to add your source files to the current project level.

Note: configuring `TRACK N` is more efficient than `TRACK Y` with `a=N` in the `LEVEL` parameters, since `TRACK N` will avoid the overhead of reading the tracking files. So, you may wish to use `IF/ENDIF` blocks to exclude frequently referenced but untracked files, even if you also use `a=N` in the `LEVEL` parameters.

`a=q` (Add-new field = Query) indicates that if an untracked module is checked-in, TLIB will query the user about whether or not to add the module to the current project level tracking file. The question asked is,

```
    Module filename.ext was previously untracked.  Do you wish t
o add it to project level name (y/n)?
```

"`a=Q`" is the default, unless your current work directory is also the current project level reference directory.

Note: If your current work directory is the current project level reference directory, then the `s=` field is ignored, and TLIB behaves as if `a=y` and `s=new` were both configured.

`s=new` (Store-to field = New) indicates that if a module was found in an "inherits-from" (`i=`) level, then when the module is checked-in with the U (update) command, it will be added to the current ("new") level.

Note that this is always what TLIB does with modules found in a `p=` (promote-to-and-inherit-from) level, regardless of the `s=` field.

Also, note that if you are using `LOCKING B` (branch/level locking), then you should also configure `s=new` on the `LEVEL` configuration parameters, since TLIB will only lock the current level.

`s=old` (Store-to field = Old) indicates that if a module was found in an "inherits-from" (`i=`) level, then when the module is stored the new version will be recorded in that level (rather than the current level). This is typically specified for "customization" levels, where most modules are *not* customized, and the programmer would like changes to those modules to be stored in the "standard" levels. This field should not be specified unless you have also specified an inherits-from (`i=`) level: this field does not affect a "promote-to" (`p=`) level.

`s=Q` (Store-to field = Query) indicates that TLIB should ask the user to decide whether to add an "inherited" module from an inherits-from (`i=`) level into the current ("new") project level, or, instead, put it back into the original ("old") project level in which it was found. The question asked is usually:

```
      Do you wish to add module FILENAME.EXT to project level
      NEWNAME?  Choose "Y" for yes; or choose "N" to update t
he
      entry in project level OLDNAME; or press ESC to enter i
t
      in neither. (y/n/Esc)?
```

However, if the correct version number is already listed in the `i=` level for this source file, then TLIB just asks whether you wish to add it to the current level. This usually happens when the U command reports that there were "no changes" in your source file, so that a new version is not created.

`s=Q` is the default (but if your current work directory is also the current project level reference directory, TLIB behaves as if `s=New`, regardless of how you've configured it).

`s=C` (Store-to field = query-if-Changed) is a "smarter" variation on `s=Q`, in which TLIB tries to avoid asking you questions if it seems obvious how you would probably answer.

`s=C` is just like `s=Q` when you are storing a new version of your source file with the U (update) command.

However, if there were no changes to the source file, so that the U command doesn't store a new version, and the version number in the predecessor (`i=`) level is already up-to-date for this source file, then TLIB will not ask you whether you wish to add the source file to the current lev-el.

Instead, what TLIB does in this case is determined by the `a=` field. If `a=N` or `a=Q` is configured (the usual case), the source file will not be added to the current level. If `a=Y` is configured, TLIB will go ahead and add the source file to the current project level, but only if `f=Y` is configured.

Note #1: If your current work directory is the current project level reference directory, then the `s=` field is ignored, and TLIB behaves as if `s=new` and `a=y` were configured.

Note #2: configuring `s=Old` or `s=Q` should also affect the operation of branch locking ("`LOCKING B`") when a module is E(xtracted) from an `i=` (inherits-from) level, since TLIB should lock the module at that level instead of the current level. However, this is not implemented in TLIB 5.50. Instead, TLIB 5.50 always locks the current level when `LOCKING B` is configured, so you should always configure `s=New` on your `LEVEL` parameters if you use `LOCKING B`.

`b=1` (Branch = 1-deep) indicates that this project level contains customizations which should not normally be stored as "trunk" (integer) version numbers. You will probably also want to specify `c=`*nn*, to specify a characteristic branch number for each level at which you've configured `b=1`; see below.

`b=0` (Branch = 0-deep) is the default, which allows new versions to be stored as trunk (integer or *major:minor*) version numbers where possible.

`c=`*mm* (Created branch number preference) You can configure "`c=`*nn*" (where *nn* is an integer) to tell TLIB what branch number you prefer be used when TLIB is creating a new branch version. The "branch number" is the (parenthesized) number of the branch, not the number of the version within the branch.

If TLIB creates a new branch to store a new version of any file for this project level, TLIB will try to use nnn as the branch number (or, if branch *nn* already exists, TLIB will create the next-higher unused branch version number).

This is for aesthetics; by using the `c=` option, you can make all the branch versions for a particular level share the same branch number (though the branches may still sprout from different trunk versions).

Suppose, for example, that you use the `U` command to store a revision to version 6 of `FOO.C`, but version 7 of `FOO.C` already exists. Then TLIB must create a branch version from version 6. If, in the `LEVEL` parameter for the current project level you have specified `c=15`, then TLIB will create the new branch version as “`6.(15)1`”.

You may want to use `c=`*nn* in combination with the `b=1` option (which forces branch creation).

For example, suppose that TLIB.CFG contains:

```
  LEVEL n=STD d=H:\TLIBLEVS\STD\ a=Y
  LEVEL n=CUST1 d=H:\TLIBLEVS\CUST1\ i=STD b=1 C=10
  LEVEL n=CUST2 d=H:\TLIBLEVS\CUST2\ i=STD b=1 c=20
```

If the current version of `BAR.C` is `7`, and you customize it for level `CUST1`, then the new version created by the `U` (update) command would (by default) be `7.(10)1`, instead of `7.1` [`7.1` is equivalent to `7.(1)1`, since the default branch number is `1`].

`w=N` (Non-Writable level) Configure “`w=N`” to tell TLIB that this is a “non-writable” project level, at which the U (update) and E (check-out for modification) commands are prohibited. This is provided for the convenience of users who must comply with ISO 9001, which mandates a 3-level promote structure, and prohibits direct updates to the upper levels. Under the ISO 9001 scheme, all work must be done at the lowest project level, and then promoted to the upper levels with the `AP` command.

`w=Y` (Writable level) This is the normal setting and the default.

Here's an example of how you might use “`w=N`”:

`TLIB.CFG` contains:

```
  LEVEL n=REL d=H:\TLIBLEVS\REL\ w=N
  LEVEL n=TEST d=H:\TLIBLEVS\TEST\ p=REL w=N f=Y
  LEVEL n=DEV d=H:\TLIBLEVS\DEV\ p=TEST i=REL f=Y a=Y
```

In this example, `DEV` (the development level) is the “lowest” level (with the highest version numbers), which is where the development work is done. All Updates must be done at the `DEV` level. It promotes to `TEST` and inherits from `REL`.
The “upper” levels (`REL` and `TEST`) are non-writable, because “`w=N`” had been configured.

The “f=Y” used in this example means that the levels are “fully-populated” (not sparse). For details about why you might prefer sparse vs. fully-populated levels see p. 171.

Here is an example of seven `LEVEL` statements from a TLIB configuration file which defines a fairly complex set of seven project levels:

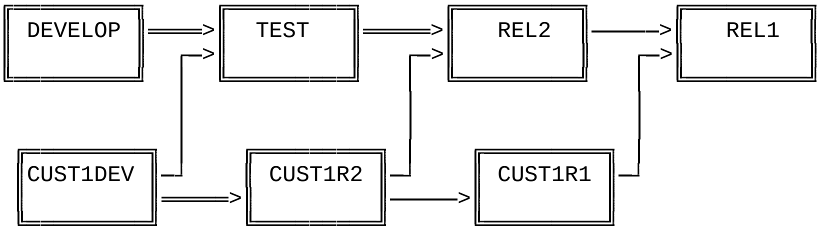
```
 ! Define 3-level promote chain (DEVELOP -> TEST -> REL2)
 ! for the “regular” version, which also “chains-back”
 ! (inherits from) the previous release (REL1):
 LEVEL n=DEVELOP d=f:\stddv\ p=TEST i=REL2,REL1 s=NEW
 LEVEL n=TEST d=f:\test\ p=REL2 i=REL1 s=NEW
 LEVEL n=REL2 d=f:\release2\ i=REL1 s=NEW

 ! Define the previous (now stable) release level, REL1:
 LEVEL n=REL1 d=f:\release1\

 ! Define a 2-level “promote-chain” (CUST1DEV -> CUST1R2)
 ! for a customized variant of the regular version:
 LEVEL n=CUST1DEV d=f:\c1dv\ p=CUST1R2 i=CUST1R1,TEST,REL2,REL1 b=1 c=20
 LEVEL n=CUST1R2 d=f:\c1r2\ i=CUST1R1,REL2,REL1 b=1 c=25 s=NEW

 ! The previous release (CUST1R1) of the customized variant:
 LEVEL n=CUST1R1 d=f:\cust1r1\ i=REL1 b=1 c=15 s=NEW
```

Here's a drawing of the relationships between the project levels defined in the above example, with the “oldest” (most stable) levels on the right:



In this drawing, the boxes indicate the seven project levels, the double horizontal arrows indicate `p=` (promotes-to-and-inherits-from) relationships, and the single arrows indicate `i=` (only-inherits-from) relationships.

This diagram is inexact, since it does not show the complete list of `i=` (inherits-from) library levels which TLIB searches when looking for a module. However, these would normally be defined in “chains.” For instance, if level `DEVELOP` inherits from `TEST`, and `TEST` inherits from

REL2, then DEVELOP would normally be configured to inherit from REL2 (as a “second choice”) as well as from TEST.

The actual levels & the order in which they are examined by TLIB is determined by the p= and i= fields, and the order in which you have specified the levels in the i= field.

Promote chains

Promote chains implement a classic “staging” approach to software project management, with successively “higher assurance levels” containing successively more rigorously-tested (but older) versions. Consider the following promote chain:

```
DEVELOPMENT ===> TEST ===> RELEASE1
```

In this example, a new version of a module would be first stored in the DEVELOPMENT level, then it is promoted to the TEST level, and then promoted to the RELEASE1 level.

Note that the “highest” level (RELEASE1, in our example) contains the oldest modules and, consequently, the lowest version numbers!

We would configure the three levels like this:

```
LEVEL n=development d=d:\dev\ p=test i=release1
LEVEL n=test d=d:\test\ p=release1
LEVEL n=release1 d=d:\rel1\
```

When programmers are working on “rough” code, they would set PROJLEV to DEVELOPMENT. This could be either by a special TLIB configuration file or by configuring “PROJLEV %!!CURRENT%”, and requiring the programmers to “SET CURRENT=DEVELOPMENT” before running TLIB (or, more conveniently, put “SET CURRENT=DEVELOPMENT” in their autoset file).

When a programmer is satisfied that his changes are correct, he uses the AP command to “promote” the changed modules to the next level (level TEST in our example).

When someone responsible for testing the modules at the TEST level is working, they would set PROJLEV to TEST. When they are satisfied that the modules are adequately tested, they promote them to the next level (level RELEASE1 in our example).

Note that there is nothing magical about the names DEVELOPMENT, TEST and RELEASE1, nor about having three assurance levels. You can name your levels anything you wish, and you can have as many or as few of them as you wish, within reason. (But if you have a promote chain ten levels deep, you should probably reexamine your strategy!)

The evolution of LEVEL structures

When the RELEASE1 level of code is actually shipped, development begins on the next release. At this time, all modules (source files) have been promoted to the RELEASE1 level.

The goal is to develop a new, improved "RELEASE2" level of code. Some of the modules will be the same as in RELEASE1, but some will be new or changed.

The plan is to use the old DEVELOPMENT and TEST levels for the new code. Modules will be promoted from the DEVELOPMENT level to the TEST level, as before, but then they will be promoted from the TEST level to the RELEASE2 level (instead of the RELEASE1 level). So, the new RELEASE2 level must be inserted in the chain "above" the TEST level (so that promotes work correctly) but "below" the RELEASE1 level (so that RELEASE2 can "inherit" the unchanged modules from RELEASE1).

The new chain looks like this:

```
          DEVELOPMENT ===> TEST ===> RELEASE2 ---> RELEASE1
```

The single-line arrow connecting RELEASE2 to RELEASE1 indicates that you cannot actually promote modules from RELEASE2 to RELEASE1 (because RELEASE1 has been shipped and is stable), but that RELEASE2 "inherits" some unchanged modules from RELEASE1.

We would change the configuration to look like this:

```
    LEVEL n=development d=d:\dev\ p=test i=release2,release1
    LEVEL n=test d=d:\test\ p=release2 i=release1
    LEVEL n=release2 d=d:\rel2\ i=release1 s=new
    LEVEL n=release1 d=d:\rel1\
```

(Note that the order in which the four `LEVEL` configuration parameters are placed in the TLIB configuration file is of no consequence.)

In this example, the module first exists on the `RELEASE1` level. Then it is checked-out, modified and stored (with the TLIB “U” command) in the `DEVELOPMENT` level. Then, when the programmer is satisfied with the module, it is promoted to the `TEST` level.

When testing is complete, the module is promoted again, to the `RELEASE2` level, where it stays.

A Larger Example

Other uses for “linked” levels are also possible. For instance, in the following diagram, double arrows indicate a promote chain, and single arrows indicate “inherits-from” linkages used for customizations:

```
DEVELOPMENT ═══> TEST ════╗
                          ╚═══>
                          ┌───> STDREL
CUST1DEV ═══> CUST1REL ───┘ ┌─>
                            │
                            │
CUST2DEV ═══> CUST2REL ─────┘
```

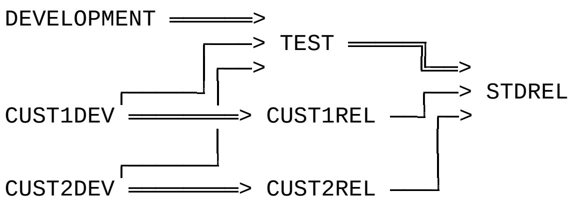
In this example, there are three assurance levels for a “standard” version of the software, plus two customizations of the standard version, each of which has two assurance levels.

We would configure the seven levels like this:

```
LEVEL n=development d=d:\dev\ p=test i=stdrel
LEVEL n=test d\d:\test\ p=stdrel
LEVEL n=stdrel d=d:\std1\
LEVEL n=cust1dev d=d:\cust1\ p=cust1rel i=stdrel b=1 c=10
LEVEL n=cust1rel d=d:\c1r1\ i=stdrel b=1 c=15 s=new
LEVEL n=cust2dev d=d:\cust2\ p=cust2rel i=stdrel b=1 c=20
LEVEL n=cust2rel d=d:\c2r1\ i=stdrel b=1 c=25 s=new
```

Another Example

In the following diagram, double-width arrows indicate a promote chain, and single-width arrows indicate customizations:



As in the previous example, there are three assurance levels for a "standard" version of the software, plus two customizations of the standard version, each of which has two assurance levels. However, in this variant, the chaining is slightly more complex.

```
LEVEL n=development d=d:\dev\ p=test i=stdrel
LEVEL n=test d=d:\test\ p=stdrel
LEVEL n=stdrel d=d:\std1\
LEVEL n=cust1dev d=d:\cust1\ p=cust1rel i=test,stdrel b=1 c=10
LEVEL n=cust1rel d=d:\c1r1\ i=stdrel b=1 c=15 s=new
LEVEL n=cust2dev d=d:\cust2\ p=cust2rel i=test,stdrel b=1 c=20
LEVEL n=cust2rel d=d:\c2r1\ i=stdrel b=1 c=25 s=new
```

How does this differ, functionally, from the previous example?

In this example, the customized "development levels" (`CUST1DEV` and `CUST2DEV`) derive from a less stable level of the standard version (the `TEST` level). Thus, if `CUST1DEV` is the current project level and you extract a module which has not previously been customized (that is, it is not listed in `CUST1DEV` or `CUST1REL`), then you will get the version currently in the `TEST` level (if it is not in the `TEST` level, it will be retrieved from the `STDREL` level, as before). If the chaining were set up as in the previous example, and you extracted a not-yet-customized module, then you would get the version listed in `STDREL`, rather than `TEST`.

Administering Multiple Project Levels

It may not always be obvious which project levels should be branches, and what you should do when you need another project level. This section is intended to advise you.

Scenario #1a:
Development level + Release level
For small to medium-sized projects
(Sparse REL2)

You started your project with just one project level, called `REL1`, which contains the trunk versions of your preliminary specifications and source code while it is under development. Now, however, the code is working, and you are ready to “release” what you have, either to your customers or to an internal Test Department for further testing. As bug reports come in, the development team will have to give priority to fixing the problems. In the meantime, however, they will begin development of the next major release.

Recommendation:

First, be sure you've updated all the TLIB libraries with the latest level of code (checked-in all modules).

Since this is release time for `REL1`, you need to use the S command to create a “snapshot” version label file which records the revision numbers for each source file at this point in time - otherwise, it will not be easy for you to recreate this version in the future. (Actually, since you have only one level, you could simply save a copy of the tracking file, instead of using the S command.)

Hint: you may wish to store the version label file into a TLIB library of its own.

Now you need to “split” `REL1`. That is, you need two project levels: one to track fixes to the new release, and the other to track the new development work for the next major release.

Change your TLIB configuration file to describe two levels instead of one:

Old:

```
LEVEL n=REL1 d=f:\rel1\ a=Y r=Y
```

New:

```
LEVEL n=REL1 d=f:\rel1\ r=Y b=1 c=5
 LEVEL n=REL2 d=f:\trunks\ a=Y r=Y i=REL1 s=new
```

Initially, the two levels are effectively the same: `REL1` will be a fully-populated more-or-less-stable level which contains the released version, perhaps with bug fixes. `REL2` will be the “rough” level where the development team stores the new, incomplete enhancements which will eventually become the second release of your product.

The “main” long-term development path is the work on the next major release (`REL2`), so that should be the “trunk” level, which means that the released version (`REL1`) should now be tracked with a “branch” project level. That's why we configured `b=1` for the `REL1` level.

We can represent the inheritance chain like this:

```
        REL2 —————> REL1
```

Initially, `REL1` and `REL2` effectively contain the same versions of each module (since `REL2` inherits all “missing” source files from `REL1`).

When you make modifications to the modules in the `REL2` level, the new versions of the source code will be stored as trunk versions in the TLIB libraries.

However, if you make modifications to modules in the `REL1` level (say, for fixing bugs), the new versions of the source code will be stored as branches in their TLIB libraries. This happens automatically because of the `b=1` field in the `LEVEL` parameter for `REL1`.

An Alternate Recommendation:

You can defer creation of the new `REL1` project level, if you wish.

If you think that you may not need to do maintenance on this release, then you needn't create the new `REL2` project level at this time. Just be very sure that you save a version label, and you can keep using the old `REL1` level for work on your second release (though the choice of names is obviously poor).

Then, if it later develops that you need to create a project level for maintenance of `REL1`, you can do so by creating a new level, and then initializing it with the saved version label files. For example, if your saved snapshot version label file is called `REL1V1.SNP`, then you can populate the new level with the old versions be making the new level your current project level and using the A (add/alter project level) command like this:

```
        TLIB A @REL1V1.SNP
```

Note that `REL1V21.SNP` can be either a snapshot file created with the S command, or a saved & renamed copy of the old `TLIBWORK.TRK` tracking file for `REL1`. TLIB will work equally well with either one.

Modules which are unchanged from the first release need not be part of the `REL2` level; they will be “inherited” from the old `REL1` level.

Scenario #1b:
Development level + Release level
For small to medium-sized projects
(Fully-populated REL2)

This is an alternative solution for the same scenario described above as “Scenario #1a...”.

Rather than making `REL2` inherit from `REL1`, you can simply set up `REL2` to be another fully-populated project level.

One way to do this is to use COPYTRAK as described in (D2, p. 133) to make a new, fully-populated tracking file for `REL2`. Then use POKETRAK (or a text editor) to change the `n=REL1` to `n=REL2` on line1 (be sure that you don't change the length of the line -- it must be exactly 126 characters).

Alternately, you can set up `REL2` as described in scenario #1a, above, and then use the AF command to fully-populate the new level:

```
      TLIB AF *.*
```

You can, if you wish, also convert from a fully-populated `REL2` level to a sparse `REL2` level. First, you must first add an `i=REL1` field to the `LEVEL` configuration parameter for `REL2` (if it isn't already there). Then you use the ADF command to remove from `REL2` those modules which are defined as the same version in `REL1`:

```
      TLIB ADF *.*
```

Scenario #2:
Development level + Test level + Release level
For large projects

In this scenario, you will use multiple project levels to implement different “assurance levels.”

The “lowest” assurance level in this example is called `TRUNKS`. It is similar to the `REL2` level of scenario #1, above. `TRUNKS` is used for “first try” rough code, for main-line development.

There is no guarantee that code from the `TRUNKS` level is working at any given time.

A `TEST` level will be used for modules which the developers believe are working correctly; this level is the level of code which the Test Department tests for its “system tests.”

A third level, called `RELEASE`, is used for modules which the Test Department has “blessed.”

```
      TRUNKS ===> TEST ===> RELEASE
```

(Note: there is nothing sacred about this three-level structure; you may use as many or few assurance levels as you wish (30 is the limit, but more than four or five would be very unusual.)

As in scenario #1, the trunk versions should be for main-line development.

The `TEST` level is a “promote-to” project level from `TRUNKS`. Because the main development process begins at the `TRUNKS` level, any changes made directly at the `TEST` level should be branches. Therefore, you may want to use the “`b=1`” and “`c=`*nn*” fields in the `LEVEL` configuration parameter for `TEST`.

Similarly, the `RELEASE` level is a “promote-to” level from `TEST`. You will create the `RELEASE` project level for the first time when the first `TEST` version has “passed” system test.

Questions & Answers for Scenario #2:

Q.1: When should a snapshot version label file be created?

A.1: Whenever you “ship” a version, and whenever you think you just might possibly someday have a reason to need to know what was in a level of code. When in doubt, save it! You can avoid a proliferation of saved snapshot files by storing them in a TLIB library.

Q.2: What happens when a bug is discovered in the development (`TRUNKS`) level of code?

A.2: This one is easy. The developer makes `TRUNKS` his current project level, then he uses “`TLIB E`” to extract (check-out) the defective source file(s) into his work directory (perhaps `C:\WORK\`), he fixes the bug, and he uses “`TLIB U`” to store the fix. The `TRUNKS` level's tracking file is automatically updated. Note that the current/work directory should be a private directory used just by this developer, perhaps on the `C:` drive. It should *not* be the project level's reference directory.

Q.3: What about a bug discovered in the `TEST` level of code? The bug then presumably exists in both the `TEST` level and the `TRUNKS` level.

A.3: You have two choices:

a) You can make the fix directly in the `TEST` level. In this case, you (the programmer) make `TEST` the current project level; then use “`TLIB E`” to extract (check-out) the defective source file(s); then use "`TLIB U`" to store the fixed version.

The `TEST` project level tracking file will be automatically updated. However, you'll eventually need to migrate the fix back "down" into the `TRUNKS` level using the M (migrate) command, and perhaps manually correct any "change collisions" which DIFF3 reports.)

b) If the `TRUNKS` (development) level has not incorporated any new code which would preclude its inclusion in the `TEST` level, you should probably make the fix on the `TRUNKS` level, then promote the fix to the `TEST` level with the AP command.

Q.4: What about a bug discovered in the `RELEASE` level of code? The bug may also exist in the `TEST` and `TRUNKS` levels.

A.4: You have three choices:

a) You can make the fix in the `RELEASE` level, then later migrate it down into the `TEST` and `TRUNKS` levels with the M (migrate) command. Or,

b) You can make the fix in the `TEST` level, then promote it (with the AP command) to the `RELEASE` level, and later migrate it down into the `TRUNKS` level. Or,

c) You could even make the fix in the `TRUNKS` (development) level, and then promote it to the `TEST` level, and from there to the `RELEASE` level. However, before doing this you should verify that there is nothing in the `TRUNKS` level modules which would "break" the `RELEASE` level code.

Scenario #3:
Semi-custom Software

In this scenario, you will use multiple project levels to maintain customized versions of the software.

If you're read scenarios #1 and #2, above, this one may seem simple. The `TRUNKS` level will be for your "standard" version. You will create one or more "customized" versions by creating branch project levels from the `TRUNKS` level.

When the time comes to bring a particular customized version up to the level of the standard version, use the M (migrate) command to add the `TRUNKS` level changes to the branch levels.

If, as in scenario #1 or #2, you have more than one “standard version” (of differing vintages), then you can create the customized version from any of the “standard” versions (e.g., `REL1` or `REL2` in scenario #1).

Using Load-Time Conditionals

Sometimes you may have two or more different work environments, so that you need to configure TLIB two or more ways. The `IFF` directive is ideal for this.

For example, a TLIB user we know has several different products, each of which gets slightly customized for each of perhaps as many as 50 or 60 customers. A particular developer may work on several products, and on both standard and customized variants of each.

Warning: the following description of how we set up TLIB for this customer assumes familiarity with TLIB's named project level mechanisms. If you're new to TLIB, you'll probably find it befuddling.

This customer has a work directory for each variant of each product, in a tree like this:

`c:\work\product1\std\`	“standard” directory for product 1
`c:\work\product1\cust1\`	1st customized directory for product 1
`c:\work\product1\cust2\`	2nd customized directory for product 1
etc.	
`c:\work\product2\std\`	“standard” directory for product 2
`c:\work\product2\cust1\`	1st customized directory for product 2
`c:\work\product2\cust2\`	2nd customized directory for product 2
etc.	

The goal was to configure TLIB so that the current project level, library path, etc. would all be deduced by TLIB automatically from the current work directory, all with a single `TLIB.CFG` file (to be placed in the TLIB executables directory). Here's an excerpt from the `TLIB.CFG` file:

```
! Get name of the current directory, and of the directory
! above it.  The current directory is the customer name,
! and the directory above it is the product name:
set CUST=%tlibcfg:workdir:-1%
set PROD=%tlibcfg:workdir:-2%
set REFDIR=d=\\acme\sys\levls\%PROD%\%CUST%
set STDDIR=d=\\acme\sys\levls\%PROD%\std
!  There must be an "ordinal.inc" file in each proj level
! reference directory, with a single line like this:
!       set ORD=20
! That sets the branch noumber for customized versions of
! each module stored at the level.  In effect, the ordinal
! is a customer number. We recommend that the ordinals NOT
! br adjacent integers.  Instead, space them by 5 or 10
! (e.g., use 5, 10, 15, 20, etc.)
! In "standard" (%PROD%_std) levels, the ordinal should be:
!       set ORD=1
include %REFDIR%\ordinal.inc
!   There is just 1 set of library files for all the custom
! (and std) variants of each product, of course:
path \\acme\sys\libs\%PROD%\
!   Current proj level is determined by current directory:
projlev %PROD%_%CUST%
!   Standard level (which might also be the current level):
level n=%PROD%_std d=%STDDIR%\ b=0 c=1 a=y
!   Last, if current level is a customized level, we must
! also configure the LEVEL parameter for the current level:
iff ('%CUST%' nei 'std')
 level n=%PROD%_%CUST% d=%REFDIR%\ i=%PROD%_std b=1 c=%!ORD% a=y
endif
```

For more information on load-time conditionals (`IFF/ELSE/ENDIF`) see p. 323.

EBF command: Fast-Extract (extract only changed files)

TLIB 5.50 adds support for a “fast extract” operation, intended for quickly refreshing (“freshening”) browse mode files in your working directories and reference directories. You use it by adding the “F” option to your extract commands, as in these examples:

```
        TLIB EBF @TLIBWORK.TRK or
        TLIB EBFS <wild-card-spec-or-@filelist> *
```

Read “EBF” (or, equivalently, “EFB”) as “Extract/browse/fast”. Read “EBFS... *” as “Extract/browse/fast/latest-trunk-version”.

For “fast extracts” to work, you must be using version tracking (that is, you must have configured `TRACK Y` or `TRACK M`).

If you are using check-in/out locking (e.g., `LOCKING Y`), then you'll probably want to configure `READONLYB Y` and `REPLROBR Y` to use this command to refresh browse mode files in your work directory.

This command is somewhat symmetrical with the “UF” (fast update) command which, if you work alone (and with `LOCKING N` configured), makes it easy to “freshen” your TLIB libraries by storing your latest changes.

However, unlike the UF command, the EF (fast extract) command does not work by comparing file dates. Instead, it works by setting and examining three fields, “`v=`”, “`l=`” and “`t=`”, in the tracking files (`TLIBWORK.TRK`).

Also, unlike the UF command, fast extracts are mainly used in networked, multiple-programmer shops. Such programmers normally have locking enabled, so they normally use “UO” (“update owned files”) instead of “UF” when they want to update/check-in all changed files.

Note: the old `FASTEBFT.BAT` kludge is obsolete in TLIB 5.50, thanks to major performance improvements in the EBF command.

Reference directories

A reference directory is the directory associated via the `d=` field of the `LEVEL` configuration parameter with a named project level. By default, the reference directory contains only the `TLIBWORK.TRK` file, so the name “reference directory” is, perhaps, a bit misleading.

However, if you wish, the reference directory can also contain “reference copies” of a complete set of source files for the project level (which is why we call it a reference directory). TLIB 5.50 can automatically maintain reference directories for any or all of your project levels.

If you wish for TLIB to keep up-to-date reference copies of your source files in the reference directories for one or more project levels, simply configure `r=Y` in the `LEVEL` configuration parameter for those levels.

For project levels with `r=Y` configured, TLIB will automatically copy the source file into the reference directory whenever you update the TLIB library with a new version at that project level.

Note that this feature does *not* respect the `OLDDATE Y` configuration parameter; the reference copy always gets the current date/time, so that your MAKE utility can correctly rebuild dependent modules. This is a subtle difference between using r=Y and using the EF or EBF command to implement reference directories.

See also the `REFSUBDIR` (below) and `FORCEREFR` (p. 306) configuration parameters.

The REFSUBDIR configuration parameter.

Syntax:

```
   REFSUBDIR directory-name
```

This configuration parameter can be used in combination with an `IF/ENDIF` block when you are not using `TREEDIRS Y` but you need to keep the reference copies of your include files in a different directory from the reference copies of your main source files.

Why would you need to do this? The usual reason is a problem with the directory search order used by most compilers when reading include files. Usually, when searching for an include file included by a particular main source file, the compilers first look in the directory containing the main source file.

Unfortunately, if the main source file is a reference copy, and the include file is one that you have checked-out into your personal work directory and have modified, then that search order is *wrong!* It will cause the old, unmodified version of the include file to be used instead of the one that you are trying to test! For reference directories to work properly, you *must* somehow prevent the compiler from reading the reference copy of an include file when you also have a modified version of that include file checked-out into your work directory.

Note that this same problem occurs when you have “customization” levels for the support of semi-custom software versions. The danger is that a customized include file may not be used when compiling a main source file which has not been customized.

If you are programming in C, you may be able to avoid this obnoxious compiler behavior simply by using <angle braces> instead of "quote marks" in your `#include` directives (though this does not work for all compilers).

Alternately, you can separate your include files from your main source files by configuring either `TREEDIRS Y` (if you also keep them separated into a “tree” of subdirectories in your work directories), or `REFSUBDIR` (if you keep all the files together in one work directory).

For example, if you are programming in C, you might configure:

```
LEVEL n=main d=f:\main r=Y
REM - Ref copies of header files go in F:\MAIN\INC subdir
IF *.h
    REFSUBDIR inc
ENDIF
PROJLEV main
```

Note that (despite the REMark in that example) the REFSUBDIR parameter applies to *all* of your project levels; you cannot configure a REFSUBDIR for just one project level (unless, of course, you have only one project level). Thus, for example, if you were to configure "REFSUBDIR INCL", then you'd also need to create an INCL\ subdirectory in each of your project level reference directories (except for project levels which don't have r=Y set in the LEVEL parameter).

However, there are several caveats to remember when using REFSUBDIR:

a) Do not use REFSUBDIR in combination with TREEDIRS Y (doing so will confuse TLIB).

b) REFSUBDIR will *not* solve the problem of the compiler reading the wrong include files if you "nest" your include directives (that is, if your include files include one another)! (Think about it.)

c) If you manually refresh the files in a REFSUBDIR subdirectory, be sure that you do so only when the WORKDIR directory (by default, the current directory) is the main reference directory for that project level. Otherwise, TLIB will not find and use the proper project level tracking file, and it will create a bogus work directory tracking file in your REFSUBDIR subdirectory.

There are also two special forms of the REFSUBDIR configuration parameter, (and these forms *can* be used in combination with TREEDIRS Y):

"REFSUBDIR nul" prevents storing reference copies altogether. (Think of nul as the "bit bucket").

"REFSUBDIR" (or "REFSUBDIR .") disables REFSUBDIR, just as if you had not configured it at all.

These special forms can be used, in conjunction with IF/ENDIF blocks, to enable and disable the creation of reference copies for various files. For instance, the following could be used if you wanted reference copies only of `.c` files, and not anything else:

```
REFSUBDIR nul
IF *.C
   REFSUBDIR
ENDIF
```

A (add/alter project level) and AP (promote) commands

These commands are used with TLIB 5.5x's named project levels (a.k.a., advanced version tracking).

Most of these commands are working in TLIB 5.54. However, two commands marked with asterisks do not yet work; they are planned for a future version of TLIB.

The "`A`..." commands are:

```
 A command:  Add-to/Alter a project level or file list
 Legal suffixes are [D,F,P,S,U,X] (plus search mode suffixes):
   A or A0 -  Add files to the current project level
   AP      -  (with Promote suffix: add to promote (p=) level)
   AS      -  (with Specify-version suffix: specify version number)
   AU*     -  (with Undo suffix: go back to previous version number)
   AX      -  (with eXclude suffix: mark files as excluded, "v=X")
   AD      -  (with Delete suffix: delete from current project level)
   AF      -  (with Fast suffix: populate a sparse project level)
 Combinations:
   APU*    -  (undo promote)
   ADF     -  (depopulate (make sparse) a project level)
   APX     -  (mark eXcluded file as eXcluded in the promote level)
 ("*" means commands planned for a future version of TLIB)
```

Note #1: Default wild-card search mode is `A` (all project levels) except for `AP` and AD, for which the default search mode is `T` (this level).

Note #2: If `PROJLEV` is not configured, these commands won't work.

The A (add/alter project level) command

The `A` command is used to add modules to the current project level. This is mainly for those who have configured `a=N` in their `LEVEL` configuration parameter, so that modules are not automatically added to the current project level by the N (new library) and U (update library) commands.

Note: it is frequently useful to specifiy the `W` (workfiles) search mode suffix with the `A` command (i.e., make it the `AW` command).

The AS (add/alter project level, specifying version) command

The AS command is just like the A command, except that it allows you to override ("specify") the default choice for the initial version number that will be recorded in the project level tracking file for the added modules.

Note that if you have configured TLIB for automatic reference directory refresh ("`r=Y`") in the current project level, then the A and AS commands will also cause an automatic extract of the source file(s) into the reference directory.

The AX (eXclude source file) command

The AX command is for handling "obsolete" source files, for which there are TLIB libraries, but which are no longer used in your program. Use the AX command to mark a module as "excluded" (obsolete) in the current project level (though it may still be in use in other project levels).

Under most circumstances, the E (extract) command will not extract excluded files that are specified by wild-cards. To make the E command find and extract the files anyhow, use the S suffix to specify explicit version numbers (e.g., `*` for latest trunk versions).

Note that if you have configured TLIB for automatic reference directory refresh, then the AX command will also (in most cases) erase the reference copy of the source file from the reference directory. (Implementation note: eXcluded source files are indicated by `v=X` in the tracking file.)

The AD (delete from project level) command

To remove a module from the current project level, use the AD (delete) command.

The AD (delete) command is similar to the AX command only for a project level which has no ‘parent’ (i= or p=) levels. If the current level has a parent level, then the two commands have quite different effects.

The AD command effectively “undoes” the A command. Thus, after you use the AD command to delete a module from the current project level, the current version of that module will be determined by the promote or inherits-from levels.

This is not true of the AX (exclude) command. The AX command is used to indicate that a module is obsolete (not used at all) at the current project level. Thus, the AX command effectively “blocks” the module from TLIB's view, even if the module is still listed in a parent level. Thus, the “A” (all levels) wild-card search mode will not find an excluded source file, regardless of whether the source file is listed in other levels.

The AD (delete) command can only be done for source files that are listed in the current project level. The AX command, however, can be done for any source file.

The AP (promote) command

The AP command adds or changes a module in the current “promote” level, as determined by the `p=` field of the current project level's `LEVEL` configuration parameter. That is, it “promotes” a module.

If you have configured TLIB for automatic reference directory refresh (`r=Y`) in the promote level, then the AP command will also cause an automatic extract of the source file(s) into the promote level's reference directory.

Note: the AP command cannot be used to promote a module which is locked (checked-out for modification) at the promote-to level. Furthermore, if `LOCKING Y` is configured, then a module which is locked at *any* level is effectively locked at *all* levels. Therefore, if you want to be able to promote a module which someone has checked-out and locked, then you should configure `LOCKING B` (branch locking) instead of `LOCKING Y`.

The APX (promote-exclude) command

To promote an ‘excluded’ (obsolete) module, use the APX (promote-exclude) command. This will mark the file as excluded in the promote level instead of the current level (or, if “`LEVEL ... f=Y`” is configured, the APX command will mark the module as excluded in the promote level as well as the current level).

Note that a source file must already be marked as excluded in the current level (via the AX command) before you can use the APX command to mark it as excluded in the promote level.

Full vs. Sparse project levels:

The behavior of the AP (promote) command is affected by the “`f=`” (“full” vs. “sparse”) field of the `LEVEL` configuration parameter.

Specify “`f=Y`” for levels which will contain the full set of source modules for your project. Specify “`f=N`” for “sparse” levels, which only contain modules that are different from those in the parent (“`p=`” and/or “`i=`”) levels.

If “`f=Y`” (full) for the current level, then after you promote a source file it will be defined in both the current level and the promote level (with the same version number, of course). That is, the AP (promote) command *copies* the module into the promote level.

If “`f=N`” (sparse), then after you promote a source file, it will be defined in only the promote level. That is, the AP (promote) command *moves* the module into the promote level, deleting it from the current level. This is the default.

The AF (make-level-full) command

To fully-populate a sparse project level, by adding to it all source files which are listed in parent levels but not in the current level, use the AF command, like this:

```
        TLIB AF *.*
```

The F suffix, in this context, “filters out” all files which are already listed in the current project level, so that those files will not be affected by the command.

You would normally use the AF command after first adding the `f=Y` (full) field to the `LEVEL` configuration parameter for your current project level. However, the operation of the AF command is not affected by the `f=` field.

Note: You can also use the F suffix when adding source files to a project level with the A command, if you also specify an appropriate wild-card search mode suffix (e.g., W, for workfiles, or L for library files). For example, if you wanted to add `.c` and `.h` files in the current directory to the current project level (skipping those which are already listed in the current project level), you could use the following command:

```
        TLIB AFW *.c,*.h
```

The ADF (make-level-sparse) command

The reverse of the AF command is the ADF command. It makes a project level as sparse as possible, by removing from it any source file which is listed as having the same version numbers that it has in a parent level. To make the current project level as sparse as possible, use the ADF command like this:

```
        TLIB ADF *.*
```

The F suffix, when used in combination with the D suffix, “filters out” all files which have different version numbers in the parent levels as compared to the current level.

You would normally use the ADF command after first changing `f=Y` to `f=N` (sparse) in the `LEVEL` configuration parameter for your current project level. However, the operation of the AF and ADF commands is not affected by the `f=` field.

Locking:

If locking is enabled, then you may not Add/Alter a module's entry in a project level tracking file with the A, AX, AP family of commands when someone else has the module checked-out/locked at that level, except for `LOCKING W` (weak locking) mode.

This is a good reason to use `LOCKING B` instead of `LOCKING Y`.

Also, if you have the module checked-out/locked, or if someone else has it locked at a different level in LOCKING B (branch/level locking) mode, then a "Note:" message will be displayed to that effect.

EXAMPLE #1: Defining a 3-level hierarchy, and using "AP" (promote)

Suppose that you have decided upon a 3-level staging scheme for the program that you are developing. The old, already shipped versions are in level REL1 (which probably will not change). The prospective next release versions are in level TEST, where your Quality Assurance (QA) department is pounding on them to flush out bugs that the developers may have overlooked. The "rough" versions which the developers are working on are in level DEVL.

The TLIB configuration looks something like this:

```
        locking b
        level n=rel1 d=f:\rel1\ r=Y
        level n=test d=f:\test\ i=rel1 r=Y s=new
        level n=devl d=f:\devl\ p=test i=rel1 r=Y
        workdir c:\work\
        projlev %!!PROJECT%
```

"Projlev devl" is appropriate for developers working on the next release, but "projlev test" is appropriate for the QA department, so in this example we let an environment variable set the current project level. The developers would use SET PROJECT=DEVL, and members of the QA department would use SET PROJECT=TEST. The SET command could be in either your DOS AUTOEXEC.BAT file (or CONFIG.SYS for OS/2), or in TLIB's AUTOSET.BAT file (AUTOSET.CMD for OS/2).

Note that the order of the LEVEL configuration parameters in your TLIB configuration file is inconsequential.

Since there are reference directories for each level, and since r=Y is configured for each level, TLIB will maintain "reference copies" of the source files for each level. (However, we do not recommend using the reference copies for purposes which could keep the files open for more than a few seconds, since this could prevent TLIB from properly updating them; for this reason, letting a source-level debugger use the reference copies is generally a bad idea.) Someone who is responsible for supporting the existing code which customers are using would use the code at the REL1 level, but someone who was responsible for testing the next release would work with the TEST level code.

Now, suppose that a particular module, XYZ.C, is at version 12 in level REL1, and is not defined at the TEST or DEVL levels (because it has not been changed for the new, upcoming release). However, suppose that you, a programmer working on the next release, identify a change that must be made in XYZ.C to support a new feature.

Your current project level is DEVL, so when you check-out/lock the module, TLIB looks first in F:\DEVL\TLIBWORK.TRK to determine the version needed. Since XYZ.C isn't listed there, TLIB next looks in the promote level, which is TEST (because of the p=test field in the LEVEL configuration parameter for level DEVL). Since the module isn't defined there, either, TLIB looks at the REL1 level (because of the i=rel1 field in the LEVEL configuration parameter for level DEVL).

Note that the levels do not automatically “chain.” That is, the levels that TLIB consults are determined solely by the p= and i= fields for the current project level (DEVL). Since REL1 and TEST are not the current level, their p= and/or i= fields have no effect.

TLIB records in the work directory tracking file, C:\WORK\TLIBWORK.TRK, the version number of XYZ.C which you have checked-out.

Now, you make your changes to XYZ.C and update the TLIB library with a new version, number 13. The current versions are:

```
         DEVL      TEST      REL1
 XYZ.C   v=13                v=12
```

However, in the course of your debugging, you find and fix several bugs in your changes, storing the new versions each time. Finally, you decide that you've gotten it right. The current versions are:

```
         DEVL      TEST      REL1
 XYZ.C   v=16                v=12
```

Because you've finished your modification to XYZ.C, and are satisfied that it is correct, you can now “promote” it to the TEST level, thus making the new version available to the QA department.

The TLIB command to promote XYZ.C to TEST is:

```
      TLIB AP XYZ.C
```

Now, the current versions are:

```
         DEVL     TEST    REL1
 XYZ.C            v=16    v=12    (sparse, "LEVEL n=DEVL f=N ...")
or
 XYZ.C   v=16     v=16    v=12    (full, "LEVEL n=DEVL f=Y ...")
```

At this point, you go on to work on other things, while someone else tests the next release, at level `TEST`. When he identifies a problem in `XYZ.C`, he tells you about it, so you again check-out the module with the E command, fix it, and store it with the U command.

Now, the current versions are:

```
         DEVL      TEST      REL1
 XYZ.C   v=17      v=16      v=12
```

Then you can promote it to the `TEST` level, just as you did before:

```
      TLIB AP XYZ.C
```

Now, the current versions are:

```
        DEVL     TEST      REL1
 XYZ.C           v=17      v=12    (sparse, "LEVEL n=DEVL f=N...")
or
 XYZ.C  v=17     v=17      v=12    (full, "LEVEL n=DEVL f=Y...")
```

The use of multiple project levels allows developers to work with the very latest "rough" versions, storing as many rough versions as they wish, even as testers are working with more stable versions. Both the developers and the testers can work with their chosen level of code without in any way interfering with people who are supporting customers in the field who still have the old version.

EXAMPLE #2: Adding a bug-fix level

Continuing with the above scenario, suppose that a bug is reported in the REL1 version, but you do not wish to make the fix directly in the REL1 level because you will still need to have the original REL1 around for some reason (say, because some customers are still using it). What should you do?

Insert another level “between” REL1 and TEST, like this:

```
REM -- added a bug fix level, REL1FIXES
level n=rel1 d=f:\rel1\ r=Y
level n=test d=f:\test\ r=Y i=rel1fixes,rel1 s=new
level n=devl d=f:\devl\ p=test r=Y i=rel1fixes,rel1
level n=rel1fixes d=f:\rel1fix\ r=Y i=rel1 s=new
```

We had to make two changes:

1) We defined a LEVEL configuration parameter for the new REL1FIXES level.

2) We added the new REL1FIXES level to the i=*names* lists of DEVL and TEST. Since we want DEVL and TEST to “inherit” the changes from REL1-FIXES in preference to the old REL1 versions, we listed REL1FIXES ahead of REL1 in the i=*names* lists for DEVL and TEST.

Now the TEST and DEVL levels will automatically inherit the fixes that you make in REL1FIXES, so long as the fixes are in modules that are not explicitly listed in the DEVL and TEST tracking files.

For modules that are listed in DEVL or TEST, perhaps because they have changed, you can use the M (migrate) command to add the fixes to DEVL and TEST (the M command utilizes DIFF3 where necessary to merge changes).

EXAMPLE #3: Adding a customization level

Suppose that, as with example #2, you need to make changes to the old REL1 level of your program after you've already started making changes (in the DEVL and TEST levels) for the next major release. However, the changes are not bug fixes. Instead, you wish to build a customized variant of REL1 (perhaps for sale to Acme Corp., an important customer), and you don't want the changes to be “inherited” by the new TEST and DEVL levels.

This scenario is similar to example #2, since you need to add another level for the new version of your program, which is a modification of the version in level `REL1`, and you need to leave `REL1` unchanged. As in example #2 the new level should chain back (via its `i=` list) to the `REL1` level, so that it will “inherit” all the unchanged modules from `REL1`.

However, unlike example #2, these changes are not intended for incorporation into the next release, so the `TEST` and `DEVL` levels should not inherit from the new level.

Here, we've added the new level (called `ACME`) to the `TLIB.CFG` from example #1:

```
      REM -- added a customization level, ACME
      level n=rel1 d=f:\rel1\ r=Y
      level n=test d=f:\test\ r=Y i=rel1 s=new
      level n=devl d=f:\devl\ p=test r=Y i=rel1
      level n=acme d=f:\acme\ r=Y i=rel1 b=1 c=10 s=new
```

Note that all three of the original `LEVEL` parameters are unchanged; the only thing we did was add the new `ACME` level, and set it to chain back to the `REL1` level. Unlike example #2, we did not add the new level to the `i=` lists for `DEVL` and `TEST`.

One other difference is that we configured the “`b=1`” and “`c=`*nn*” options for the `ACME` level. While not strictly necessary, this is always a good idea for “customization” levels, since it ensures that the new versions you create for Acme will be stored as branch versions, leaving the trunk versions available for “main line” development (so that the modules in `DEVL` and `TEST` will contain trunk versions).

EXAMPLE #4: Using both bug-fix and customization levels

Suppose that we had a bug-fix level, as in example #2, and we wished to add a customization level, as in example #3. We start with the levels defined in example #2, and add a new level (`ACME`), just as in example #3, except that it is probably appropriate to make the `ACME` level inherit the fixes from `REL1FIXES`, like the `DEVL` and `TEST` levels do:

```
  level n=rel1 d=f:\rel1\ r=Y
  level n=test d=f:\test\ r=Y i=rel1fixes,rel1 s=new
  level n=devl d=f:\devl\ p=test r=Y i=rel1fixes,rel1
  level n=rel1fixes d=f:\rel1fix\ r=Y i=rel1 s=new
  level n=acme d=f:\acme\ r=Y i=re1lfixes,rel1 b=1 c=10 s=new
```

M command: Migrate changes

With the “M” (migrate) command, you can now migrate (merge) changes from one entire project level into another, all at once.

The M command analyzes the revision histories for each source file to determine which changes have already been migrated, and it uses DIFF3 (where necessary) to merge the new changes. In most cases, the only manual task that you are left with is reconciling any “change collisions” which DIFF3 may have flagged.

You will usually use the M (migrate) command with wild-cards and TLIB's project level version tracking to migrate changes from one project level into another.

Syntax:

```
TLIB M files <versions-to-be-merged>
TLIB MS files <target-versions> <versions-to-be-merged>
TLIB MF files <versions-to-be-merged>
TLIB MFS files <target-versions> <versions-to-be-merged>
TLIB MSS file <target-version> <base-version> <version-to-be-merged>
TLIB MFSS file <target-version> <base-version> <version-to-be-merged>
```

Note: The “`F`” (“fast”) suffix just makes the `M` command quietly skip already-migrated files, for a more concise `MIGRATE2.BAT` file.

EXAMPLE #1 (third-party library):

Last year, you bought a “third-party library” in C-language source code to do windowed user interfaces, ISAM file access, etc..

Prudently, you stored the original source modules, as received from the vendor, into TLIB libraries (as version 1 of each source file).

Then you modified the source code to make it work better with your application. Of course, your new versions are also stored in the TLIB libraries, as later trunk versions.

Now, after you've made many modifications to the original version, you've received a new version from the vendor. Of course, the new version (you are told) is enormously improved in a dozen ways, and fixes several subtle (but potentially catastrophic) bugs that were in the original version... and the vendor no longer supports the old version.

So, you are now faced with having to merge your own modifications with the vendor's. Big job, right?

No! Simply store the new versions in the TLIB libraries (probably as "branches," e.g., version "`1.*`"), and then TLIB can migrate the changes for you, all at once, with the M (migrate) command. See p. 181 for details.

EXAMPLE #2 (semi-custom software):

You develop and sell a semi-custom software product, written in Clipper, for professional practice management. You call your product Practice Management Software (PMS for short).

Your standard version is tracked in level `STD`, with tracking file `c:\std\tlibwork.trk`. However, you generally have to modify it to meet the needs of your customers. Whenever you do so, you create a project level for that customer.

As a service to those customers who pay an "extended support" fee, you provide them with monthly updates to their customized version of the software, incorporating fixes for all known bugs, and sometimes other improvements. So, every month you must migrate your latest improvements from the `STD` level into each of the customization levels.

Does this sound like a monthly nightmare? It's not!

For instance, suppose that last month you customized your then-latest version of the software for Dr. John Smith, DDS. You built Dr. Smith's version in level `SMITH`, with tracking file `c:\smith\tlibwork.trk`.

Now, it is the first of the month, and you have fixed several bugs in the standard version of the software, so now you must bring Dr. Smith's version up to the latest level, but without losing his customizations.

No problem! Simply go to `c:\smith` (or a suitable work directory) and use the M (migrate) command to merge the changes from level `STD` into level `SMITH`. Then recompile, run your standard suite of regression tests, and mail the diskette to Dr. Smith. See p. 181 for details.

Note: if you work alone, with locking disabled, then you can do your work in the `SMITH` level's reference directory. However, if there may be other programmers working on Dr. Smith's version, you should enable locking (configure `LOCKING Y` or `LOCKING B`) and do your work in a private work directory, but with `PROJLEV` set to `SMITH`.

EXAMPLE #3 (merging fixes into the next release):

Last January, your company shipped “release 1” of your software product, written in Pascal. At that time, you had only one project level, `REL1`.

When `REL1` shipped, you created a new project level, `REL2`, and your development team immediately began work on “release 2.” Simultaneously, your change team continued to make occasional bug fixes and other minor improvements to the original release, at level `REL1`.

Now, you need to migrate these fixes from `REL1` into `REL2`.

It's easy! Just use the M (migrate) command to do it all at once. See p. 182 for details.

Note: TLIB will (eventually) extract/check-out the needed files, so be very sure that any modifications that you've recently made to your source files have *already* been stored into the TLIB libraries. If `LOCKING Y` is configured, then use the “`TLIB UO *.*`” (update owned files) command to check-in/unlock the files BEFORE doing the M (migrate) command.

How to use the M command on the examples:

EXAMPLE #1 (third-party library):

Your latest/greatest modified versions are stored in the TLIB libraries as “trunk” (integer) versions, and the new release(s) that you get from the vendor are always stored as branches from version `1` (e.g., `1.1`, `1.2`, `1.3`, etc.).

First, make the current directory the work directory which you use for your regular (trunk) versions, which you now want modified to reflect the vendor's new modifications.

Next, make sure that any modifications that you've recently made to your source file have already been stored into the TLIB libraries. *This is important!* If locking is enabled, you can use the UO (update owned files) command to ensure this; if locking is disabled, use the UF command, instead.

Now you can migrate the vendor's latest changes into your modified source code like this:

```
TLIB M *.c,*.h 1.*
or: TLIB MS *.c,*.h * 1.*
```

Note that this simple case works even without the use of project level tracking, since you've used a version number convention to distinguish your versions from those of the vendor. Your versions are trunk version numbers (the latest is “`*`”), and the vendor's releases are branches from version `1` (the latest is “`1.*`”). However, most other scenarios do not lend themselves to this trick, and so are best handled through the use of TLIB's named project levels.

EXAMPLE #2 (semi-custom software):

You need to migrate the latest changes from level `STD` into level `SMITH`. First, make your current work directory the one which you normally use when working on Dr. Smith's customized version of your program. If you work alone, then it is probably the reference directory for level `SMITH`, c:\smith\. Otherwise, it may be a private work directory. Also, be sure that your current PROJLEV is set to SMITH (this is automatic if your current directory is the reference directory for level SMITH).

Now, you can migrate the latest changes from STD into the current level (SMITH) with the following command:

```
 TLIB MT *.prg @c:\std\tlibwork.trk
```

Here you have used the project level tracking file for level STD as if it were a version label, to tell TLIB what versions you wish to merge into the current level.

Note the use of the T ("this level") wild-card search-mode suffix to tell TLIB that you want it to search the current project level for source files (instead of finding all files for which there are TLIB libraries, which is what the default L search-mode would do).

You could also have used a snapshot version label taken (with the S command) earlier in the development history of the STD level. For instance, if you took a snapshot on July 15, 1992, and called it 92-07-15.snp, then you could migrate the STD changes through July 15, 1992 with the following command:

```
 TLIB MT *.prg @92-07-15.snp
```

If STD is a sparse project level (in which only a subset of the source files were actually listed, with the rest being listed in promote-to and/or inherit-from "parent" levels), then you should generally use a snapshot instead of using STD's tracking file directly, so that all the needed version numbers will be specified.

EXAMPLE #3 (merging fixes into the next release):

You need to migrate the fixes from REL1 into REL2. This example is handled almost exactly like example #2, except that you are merging from REL1 into REL2 instead of from STD into SMITH.

First, make your current work directory the one which you normally use when working on REL2, and be sure that your current PROJLEV is set to REL2.

Now, if the tracking file for `REL1` is `f:\rel1\tlibwork.trk`, you can migrate the latest changes from `REL1` into the current level (`REL2`) with the following command:

```
TLIB MT *.pas @f:\rel1\tlibwork.trk
```

Or, if `REL1V2.SNP` is a snapshot of `REL1`, you could use this command:

```
TLIB MT *.pas @REL1V2.SNP
```

Note that you can use all the usual TLIB wild-card search-mode and file list mechanisms to specify your source files. For instance, if you only wanted to migrate the changes for a subset of the files in the current level, you could use a file list:

```
TLIB M @files.lis @REL1V1.SNP
or: TLIB M file1.pas,file2.pas,file3.pas @REL1V1.SNP
```

Finishing the job

The M (migrate) command handles several different cases in different ways:

1) Some files will probably not need to have any changes made, because there are no changes for those modules in the versions that your are migrating into the current level.

These files are left alone.

2) Some files can simply be copied from the versions to be merged, because the current versions are base versions from which the versions to be merged were derived.

For such a file, the needed version is extracted from the TLIB library, and a special "`[COPIED...]`" supplemental comment line is added to the top, using the SCOPY program. If check-in/out locking is enabled, then the file will be left checked-out to you.

Later, when you use the U (update) command to store the new version, the supplemental comment line will be removed and used in the TLIB comments for the new version, thus recording for posterity where the version came from (so that future migrates will work correctly).

3) Some files will need to be merged with DIFF3, because they've been changed in both the current level and in the versions that you are migrating into the current level.

For such a file, three different versions of the source file are extracted (into temporary files), DIFF3 is used to do the merge operation, and a special "`[MERGED...]`" supplemental comment line is added to the top of the file.

Later, when you use the U (update) command to store the new version, the supplemental comment line will be removed and used in the TLIB comments for the new version, thus recording for posterity which versions were merged to create the new version (so that future migrates will work correctly).

If check-in/out locking is enabled, then the file will be left checked-out to you.

The M command does not actually extract the temporary files and run DIFF3 with them. Instead, the M command creates a file named `MIGRATE2.BAT`, which contains TLIB, DIFF3, and SCOPY commands to do the actual migration.

This gives you the opportunity to examine what TLIB will be doing, and intervene, if you wish.

For instance, you may need to run a "pretty-printer" (source code reformatter) on the temporary files before doing the DIFF3 merge. Such a step is not recommended unless one of the versions has been reformatted already, in which case DIFF3 would otherwise be unable to do a proper merge because of the pervasiveness of the changes.

Generally, you should first run `MIGRATE2.BAT` to do the merges, and then examine any modules for which it warns that there were "collisions" (conflicts) between the merged versions. The collisions are marked with distinctive "flag lines." (The flag lines normally containing "`###`", which you can search for with your editor, but they can be changed with the `D3-COLLIDE` configuration parameter).

If any of the merged files have been reformatted, it will be all too obvious when you look at the huge "collision areas" marked by DIFF3. For such cases, you can edit the `MIGRATE2.BAT` file to make it reformat the three input files before running DIFF3, to make them more consistent with one another (use the same pretty-printer that caused the problem, if possible). Then you can run the modified `MIGRATE2.BAT` again to retry the merge.

Other “collisions,” if there were any, must be resolved manually, by examining the flagged collision areas and correcting the problems with a text editor. Be sure to leave intact the special “`[MERGED_...]`” and “`[COPIED_...]`” comments at the beginning of the files.

Hint: To capture a “log” of everything that happened when you ran `MIGRATE2.BAT`, use Chris Dunford's wonderful “freeware” CONCOPY utility. With Mr. Dunford's kind permission, we've included a copy of CONCOPY with the shareware and public domain utilities collection on the TLIB CD.

After you've used the M command to migrate your changes, and you've run `MIGRATE2.BAT` to do any DIFF3 merges that were needed, you can delete `MIGRATE2.BAT`.

Then you should test your new versions. In our experience, if there were no collisions, the merged version almost always works correctly. However, this is not guaranteed. In fact, it seems rather surprising, since there are many ways for two different changes to be incompatible without having obvious textual collisions that are detectable by DIFF3.

For example, the two merged versions could each have added a new variable, for completely unrelated reasons, but with the same name.

Nevertheless, such cases don't seem to crop up very often.

After you have finished testing, you can use the TLIB U (update) command to store the new versions of your source files in their TLIB libraries, and the TLIB S (snapshot) command to label the new versions.

MSS and MFSS -- Overriding the “Base” Version

Occasionally you may want to override TLIB's determination of the most recent common ancestor of the two versions to be merged. For this, TLIB provides the MSS and MFSS commands.

The `MSS` (and `MFSS`) commands take four parameters:

```
 TLIB MSS filename.ext toVer baseVer fromVer
```

For example, suppose that version 7 of `myfile.prg` is the latest "standard" version, and version 7.(5)3 is a customized variant of it. Now, suppose that you fix a bug in version 7.(5)3, and store it as version 7.(5)4, but the bug was also present in the standard version.

Of course, the problem would have been simpler if you'd put the bug fix onto the standard version first, instead of the customized version. Then you could have simply migrated all improvements to date into from the standard version into the customized version. Unfortunately, sometimes you may not have the luxury of making that choice; if the customer has a "sev 1" problem, you may have no choice but to first implement the fix in his customized version.

In that case you cannot use the `M` or `MS` command to merge the bug fix into version 7, because TLIB has no way of knowing that only the last revision to this customized variant is applicable to the standard version. If you were to do `TLIB MS myfile.prg * 7.(5)4` then TLIB would add all the changes between versions 7 and 7.(5)4 to the latest trunk version, with the result that the customizations would also become part of the standard version (which is not what you want).

Instead, you should use the `MSS` command, to tell TLIB that you want only the changes between versions 7.(5)3 and 7.(5)4 added to the standard version:

```
 TLIB MSS myfile.prg * 7.(5)3 7.(5)4
```

The `MFSS` command is just like the `MSS` command, except that TLIB will silently skip files for which no action was needed. That doesn't do any good for the example above, but is can be useful when the changes must be made to many files.

Suppose, for example, that `lastweek.snp` is a snapshot taken (with the `S` command) of your customized project level before you made an elaborate set of changes to many of the source files in the customized level, to fix a complex bug. Also, assume that the standard level's project level tracking file is `f:\std\tlibwork.trk` and the customized level's project level tracking file is `f:\cus\tlibwork.trk`.

You need to migrate that bug fix into the standard project level. To do so, you would do the following command (while working at the standard level):

```
 TLIB MFSS *.* @f:\std\tlibwork.trk @lastweek.snp @f:\cus\tli
bwork.trk
```
(should be all on one line)

Or, if the standard version is always the latest trunk versions, this does the same thing:

```
TLIB MFSS *.* * @lastweek.snp @f:\cus\tlibwork.trk
```

That tells TLIB to migrate into the current (standard) level all changes made to the customization level since the `lastweek.snp` snapshot was taken.

As with all of the Migrate commands, the result is a batch script, `MIGRATE2.BAT`, which you must run to finish the migration. After you have done so, you will be left with a set of checked-out files which have had the needed changes made to them. If there were any "conflicts" which TLIB's DIFF3 utility was unable to resolve (i.e., if your fix changed lines that were specific to the customized version), then you'll have to resolve the conflicts manually. They will be flagged in the source files with distinctive text lines, “`###Change collision detected`” (or whatever you have configured for the `D3COLLIDE` configuration parameter), so that you can easily find them with TLIBSCAN or your favorite GREP utility.

A comparison between the `M`, `MS`, and `MSS` commands may be instructive. They all do the same thing, except that with the `MSS` command you must specify all three version numbers for the merge, but the `MS` and `M` commands determine one or two of the versions for you automatically.

With the `MSS` command, you specify a total of three versions, and what is to be merged into the first of them is specified by a pair of versions (*baseVer* and *fromVer*):

```
TLIB MSS filename.ext toVer baseVer fromVer
```

But with the `MS` command, you do not specifiy the base version; it is determined automatically by TLIB as the most recent common ancestor of the other two versions:

```
TLIB MS filename.ext toVer fromVer
```

The `M` command is similar to the `MS` command except that the version into which the changes are to be added is not specified, either. It is taken as being whatever version is current at the current project level:

```
TLIB M filename.ext fromVer
```

A Limitation

The M command does not handle added, deleted, excluded or renamed files. You'll have to handle such files manually.

A Note About Performance

The current TLIB 5.xx release may seem unpleasantly sluggish when you migrate changes from a snapshot or tracking file like this:

```
TLIB M *.* @snapfile.ext
```

What is happening is that TLIB is finding every file which matches "`*.*`", using the default wild-card search mode (L, all library files). Then, it looks for each file name in turn in snapfile.ext, using a dumb, brute-force search, to determine the version numbers.

Those file names that are not listed in `snapfile.ext`, TLIB skips. However, if you have a lot of files, and `snapfile.ext` only lists a few names, TLIB can spend a great deal of time searching `snapfile.ext` for missing names.

A future version of TLIB will speed this process up. However, in the meantime, you can speed things up by limiting the number of files that TLIB examines. For instance, you could build a file list like this:

```
COPYTRAK -n snapfile.ext justname.lis
TLIB M @justname.lis @snapfile.ext
```

This simply makes a file list with the names in `snapfile.ext`, and performs the M command with just those files (instead of "`*.*`").

The special TLIB “-n” and “-t” command-line options

TLIB supports two command-line options which are intended mainly for use by the M (migrate) command in the generated `MIGRATE2.BAT` file. In command-line versions of TLIB, they are supported on the command line only, not in interactive mode. However, they serve to make `MIGRATE2.BAT` more concise than it would otherwise be.

a) The `-t` (temporary file) option. `-t` is just a shorthand equivalent for:

```
   C "locking_n" C "track_n" C "readonlyb_n"
```

and:

```
   C "keyflag" C "logflag"
```

That is, it overrides those 5 configuration parameters.

b) The `-n`*filename* (override Name) option is used in conjunction with the E command to extract a source file into an alternatively-named output file. (Note: When `-n`*filename* on the command line is used, TLIB does not update anything the work directory tracking file.)

The `-t` and `-n`*filename* options are used (in the `MIGRATE2.BAT` file) by TLIB's M (migrate) command when extracting temporary files; see also p. 108.

Journal file

TLIB can create a chronological “journal file” containing information about the operations which TLIB performs. Two configuration parameters are available for using this feature:

```
JFILE <filename>
and
JOPTIONS <option-characters>
```

To create a journal file, you must specify the file name using `JFILE`. For instance, if you want the journal to be named `JOURNAL.DAT` in the root directory of the `C:` drive, you could specify:

```
JFILE C:\JOURNAL.DAT
```

The JOPTIONS configuration parameter is optional. It allows you to select the kinds of information which you wish to have stored in the journal file. There are currently seven legal option characters:

`U` - commands which update the library

`I` - commands which check-in a file (with locking enabled)

`O` - commands which check-out a file (with locking enabled)

`B` - commands which extract a file, even in browse mode

`C` - the 1st comment line

`A` - all comment lines

`P` - commands which alter a project level

The default configuration value for JOPTIONS is:

```
JOPTIONS UOCAP
```

This causes TLIB to record in the journal all commands which update the library (since `U` is specified) or modify a project level (since `P` is specified), and all check-out (locking) operations (since `O` is specified). Additionally, the comment lines associated with every update operation will be recorded (since `C` and `A` are specified).

To record only the first comment line (instead of all comments), you can remove the “`A`” option letter, configuring like this:

```
 JOPTIONS UOCP
```

Unless `A` is specified, every journal entry is one line, consisting of 13 fields separated by 12 commas. The last field is the (optional) comment line, which is surrounded by quote marks and is the only field that can contain blanks. When additional comment lines are recorded in the journal (because the `A` option is set), the first 12 fields are null (that is, the line begins with 12 consecutive commas).

This “comma-delimited” format is consistent with the input requirements of common data base programs, which can be used for monitoring and report generation from the journal file by project managers.

We've also included a sample AWK program, `JOURNAL.AWK`, to parse the journal file.

Note: Your report generator should not read the journal file directly, since while the journal file is open TLIB will be prevented from accessing it. Instead of reading the journal file directly, use the included SCOPY (shared access file copy) program, `SCOPY.EXE`, to make a copy of the journal file, and then run your report generator against the copy. For usage instructions, run SCOPY with no parameters.

The 13 fields in each journal line are:

1 date of operation (in TLIB format, `DD-MMM-YY`, e.g. `19-Oct-97`)
2 time of operation (in TLIB format, `HH:MM:SS`)
3 command (e.g., `UF` for fast-update)
locking mode:
`N` for locking disabled
4 `W` or `Y` for whole libraries locked
`B` for branch/level locking
`weak` for weak locking

`5`	current user ID
`6`	library file path
`7`	text file path
`8`	lock file path (if command did check-in or check-out)
`9`	version number
`10`	source file's “key” (name) in the tracking files
`11`	current project level
`12`	other project level used, if any (see below)
`13`	quoted comment line (1st comment line includes source file date/time)

Note: fields 10-12 are new in TLIB 5.

The journal file is a normal text file, just as text-format TLIB libraries are. If you configure “`ADDCTRLZ Y`”, the journal will end in a Ctrl-Z (ASCII 26) character. If you configure “`READONLY Y`”, the journal will be kept as a read-only file, to prevent accidental deletion (you can use the DOS “`attrib`” command to reset the read-only attribute).

Changes in TLIB 5

New fields:

The journal file now has 13 fields instead of 10. It records the project level “key” (which is just the source file name unless you've configured `TREEDIRS Y`) in the 10th field, and it records the relevant project level name(s) in the 11th and 12th fields. The old 10th field (the comments for N and U commands in TLIB 4.12) is now the 13th field.

The 11th field always indicates what the current project level was when the command was done.

The 12th field, if present, indicates what “other” project level was used. If the 12th field is absent, it means “the current project level”.

The only command which always has a 12th field is the AP (promote) command, for which the 12th field is the name of the promote level.

For the U (update) command, the presence of the 12th field indicates that the command stored a module into an “inherits-from” level (one of the level names listed in the `i=` field of the current project level's `LEVEL` configuration parameter).

For the E (extract) command, the presence of the 12th field indicates that the module was not listed in the current project level, and TLIB consulted the named level to determine the version number for the extracted source file.

JOPTIONS configuration parameter:

Journaling of the A and AP commands is enabled by including the new “`P`” option letter in your `JOPTIONS` configuration parameter.

The default is now “`JOPTIONS UOCAP`”.

U (update) with a file list

In TLIB 5.0, we changed the semantics of U (update) with a file list, so that, for example, "`tlib u @tlibwork.trk`" will now do something reasonable.

Specifically, we changed it so that "fixed" version numbers are ignored in file lists used for update commands, except that if the fixed version number is inconsistent with the version number obtained from the tracking file then a warning is generated (only when version tracking is enabled, of course).

(Recall that "fixed" version numbers are version numbers which specify an exact version, and "floating" version numbers are version numbers which end in an asterisk.)

Note that you can still specify either a fixed or floating version number on the command line or interactively if you use the S (specify-version-number) suffix with the U command (i.e., the US command).

Also, floating version numbers in file lists are still respected by the U command.

The E and N commands are unchanged: they both still respect both fixed and floating version numbers in file lists (except that the N command will not create a new library file with a non-trunk starting version number).

Whereis - file finder

WHEREIS is a handy file-finder, which (like TLIB) supports multiple & leading asterisks in wild-card specifications (even under DOS), as well as options to find hidden/system files. Under OS/2, it will also tell you which files have “extended attributes” attached, and (optionally) how big the extended attributes are.

WHEREIS 1.9 supports multiple wild-card specifications separated by commas and/or spaces. When separated by commas, the drive letter applies to all the wild-card specifications within the comma-delimited list.

Examples (note the subtle difference between them):

```
whereis cd:*.c,*.h
```
(find all *.c *and* *.h *files on* C: *and* D:*)*

```
whereis cd:*.c *.h
```
(find all *.c *files on* C: *and* D:*, and all* *.h *files on the current drive)*

WHEREIS exits with errorlevel 1 if no matching files were found; otherwise it exits with errorlevel 0.

WHEREIS supports many command-line options; run it with no parameters for detailed instructions.

Quiet mode

TLIB 5.0 added support for the `-Q` ("quiet") command line option, to suppress display of the TLIB "banner line" (with the copyright notice, etc.) as well as a few other "noise" messages which TLIB normally displays.

The `-Q` (or `-q`, or `-q1`) option, if used, should be the first thing on the TLIB command line.

You can also specify "`-q2`", which is just like the "`-q`" or "`-q1`" option except that it also suppresses display of any user-configured `BANNER` lines (see the `BANNER` and `NUMBANNER` configuration parameters, p. 333).

Actually, TLIB is not really very quiet even with the `-Q` option. The `-Q` option only makes TLIB a bit less verbose.

See also the `QUIET` configuration parameter, p. 279 , which is similar but does not suppress the TLIB copyright banner.

Redirecting standard output

An unusual feature of TLIB (command-line versions) is that it will echo error/warning messages and prompts to "standard error" (stderr) as well as "standard output" (stdout) when stdout has been redirected.

This was done, in part, because some ex-Unix users like to redirect stdout to `nul`, which caused them to miss important error messages. It also makes it possible to redirect output into a file, and still see see the prompts, etc..

To see how this works, you could do:

```
tlib2 e nonexistantfile >nul
```

Only the error message is echoed to stderr, so all you'll see is something like this:

```
ERROR: No such library file: "f:\srclibs\nonexist._$"
```

We also added the "`-e`" command-line option to TLIB, to override TLIB's automatic determination of whether stdout has been redirected.

You may specify "`-e1`" to tell TLIB that stdout has not been redirected, so that TLIB will *not* echo error/warning messages and prompts to stdout.

This is mainly for use when "piping" output to a program which then displays the output on the console, such as MORE or (under OS/2 or NT) TEE. When piping output to MORE, and when using TEE to capture the output under OS/2 or NT, you can specify "`-e1`" to avoid seeing duplicate copies of the error/warning messages and prompts. Examples:

```
   tlib2 -e1 | tee >tlib.log
or tlib2 -e1 u *.c,*.h My comment | tee >tlib.log
or tlib2 -e1 t *.c,*.h | more
```

You can also specify "`-e0`" to force TLIB to echo error/warning messages and prompts to stderr even when stdout has not been redirected, though this is less commonly used.

Poketrak & Copytrak

POKETRAK is a little utility for “poking” (changing) specific fields and records in a TLIB version tracking file.

Run it with no parameters for help.

COPYTRAK is a little utility for sorting, copying and otherwise modifying entire TLIB version tracking files.

Run it with no parameters for help.

Note that you should not replace a `TLIBWORK.TRK` file with a sorted or otherwise reordered one if anyone might be running a copy of TLIB at the time and accessing that file, since TLIB “remembers” the locations of the records in the tracking files, and does not expect them to move.

Saving Disk Space

When you are not working on a particular customized version, you can save disk space by deleting the source code. First, of course, you'll want to make sure that your TLIB libraries are fully up to date, and that the tracking file, `TLIBWORK.TRK`, contains the correct version information for each module.

A simple way to do both things in one step is to utilize TLIB's `DELETESRC` configuration parameter, like this:

```
      TLIB C "DELETESRC Y" U *.C
```

Or, you could update all the libraries with the U command and then just "`DEL *.C`" or even "`DEL *.*`". That sounds radical, but the `TLIBWORK.TRK` file is stored as a read-only file to prevent it from being deleted. Be careful, though, lest you accidentally delete something else of value (e.g., `TLIB.CFG`).

Note: the best time to do this is right after you have backed-up your hard disk. Otherwise, you are "putting all your eggs in one basket," and if a hard disk failure damages one of your TLIB libraries, you may lose your only copy of that module.

Note: you can use the DOS "`ATTRIB`" command, or TLIB's `WHEREIS.EXE` program (p. 195), to list the files and to tell which one(s) have the read-only and/or archive attributes set. For instance, "`whereis .\tlibwork.trk`" will display the attributes of the TLIB version tracking file.

Temporary Files

Note: this section discusses the temporary files that TLIB creates and uses. If you want to know how to extract temporary copies of source files from TLIB see pp. and 108.

TLIB 5.50 sometimes needs to create one or more (usually small) temporary files, usually called `$TLIB_TM.0` or `$TLIB_TM.2` (but any name from `$TLIB_TM.0` through `$TLIB_TM.9` is possible). TLIB will create these files in one of four possible places, in the following order of preference:

o Where your `TMP` environment variable, if any, indicates. For example, you could put "`SET TMP=D:\`" in your `autoexec.bat` file (or in `config.sys` under OS/2) to make TLIB put its temporary files on the `D:` drive.

o Where your `TEMP` environment variable, if any, indicates.

o In the current directory.

o If all else fails, as a last resort TLIB tries to create its temporary files in "`C:\`".

Note that `TMP` (or `TEMP`) must be a real environment variable; TLIB will not use `TMP` or `TEMP` settings from your `TLIB.CFG` or `AUTOSET.BAT` file.

Performance will be enhanced slightly if `TMP` (or `TEMP`) is set to a ramdisk (`VDISK`).

Network Bug Workarounds

TLIB has code to detect (and avoid the worst consequences of) certain potentially-catastrophic network bugs.

Some network software caches writes and immediately returns a “success” result after any attempt to write a file, before the file has actually been successfully written. If the write then fails for some reason (e.g., because of a hardware problem, a disk-full condition, or a per-user disk space allocation limit), the program writing the file may not see any indication of the failure.

That can be a *disaster!* If the file being written was a TLIB library, it is *critically* important that it be written correctly. For instance, if `DELETESRC Y` is configured, TLIB deletes the source file after finishing an update, so if the library wasn't correctly written your source code could be lost!

So, we included code in TLIB to check the size of the updated library file after it is closed, to make very sure that the update was successful. If the library file is too small, TLIB now aborts with an error message, without deleting the source file (even if `DELETESRC Y` is configured).

This almost always prevents any data loss. The one exception that we know of is a network which uses Windows-NT 3.5 and the NTFS file system on the server. That version of Windows-NT 3.5 contains a cache-coherency bug that can occasionally substitute zeros for data on the server, even though the program that wrote the data can read it back correctly. It is not possible for TLIB (or any other program) to detect and work around this bug. Fortunately, Microsoft fixed this bug with Service Pack 2 for NT 3.5. *So, never use a Windows-NT 3.5 server with NTFS unless you've applied service pack 2!*

The most common network-related problem for TLIB users has to do with file ownership issues under Novell NetWare. Furtunately, this problem is not a threat to data integrity. The problem results from the Novell facility for limiting disk usage by any one user. If you're using this facility, then you should be aware that when someone adds another version to a TLIB library, the storage used accrues to the "owner" of the TLIB library (usually the person who first created the library with the N command), regardless of who is doing the TLIB U command. Thus, you can "run out" of disk space because someone else has used up their allotment.

To determine who owns a TLIB library, use the `NDIR` command. To determine whether the owner has used up his disk allotment, log in under his user ID and use the `CHKVOL` command.

If the owner user ID is no longer defined (say, because the employee left the company) then that ID has no disk allotment, and the errors can occur regardless of how much space they were once allotted. If this is what caused the problem, then you need to use `FILER` to change the NetWare owner ID for each of the TLIB library (or journal or tracking) files.

The `READ_ME.TOO` file that comes with TLIB has up-to-date information on using TLIB with many different kinds of networks.

Listbld – file list builder

LISTBLD is a program to build and manipulate file lists. If you program in C, Pascal (Borland or Microsoft), MASM, COBOL, QuickBASIC, Fortran (Lahey or Microsoft), Intel 8096 assembler, Batch90 or DBC, then LISTBLD can create file lists automatically by scanning your source code for "include" statements (and COBOL "`COPY`" verbs). It can also use wildcards and other file lists to add and remove files from file lists, and it can augment file list entries with version numbers or branch specifications for using TLIB with tree-structured library files.

dBase and Clipper programmers can generate TLIB-compatible file lists using *dBFind*, from The Software Development Factory, P.O. Box 1106-B, Hunt Valley, MD 21030. Tel: (410) 666-8129.

If LISTBLD does not recognize include statements in the programming language you use, please contact us. We will try to add support for your programming language.

Since it does not do a full parse of the source file, LISTBLD may occasionally make mistakes when scanning for include statements. For this reason, it echoes the include statements and file names to the console as it runs; if you see that it has deduced a file name incorrectly, you can manually edit the file list to correct the mistake, or use the LISTBLD `REMOVE` option to remove the erroneous file list entry, or simply ignore and tolerate the "file not found" error messages from TLIB. The usual cause for this is "commented out" include statements (or COBOL "`COPY`" verbs).

If your compiler allows "nested" include statements (that is, include statements within include files), then LISTBLD can be instructed to scan the nested include files, too, like this:

```
LISTBLD inputfile outputfile NEST
```

Otherwise, you would run it like this:

```
LISTBLD inputfile outputfile
```

`Inputfile` is normally the “main” (outer) program source file, and `outputfile` is often “`*.lis`”.

If there are several modules in your program, you can run LISTBLD once for each of them, using the `ADD` option to make LISTBLD add each set of new names to the same file list (`outputfile`), like this:

```
LISTBLD inputfile outputfile ADD
  or
LISTBLD inputfile outputfile NEST ADD
```

Note that LISTBLD detects and deletes duplicate names, so `outputfile` will not end up with duplicate list entries even if two or more modules use the same include file.

To add file name(s) to a file list *without* scanning for include statements, you can use the ONLY option, like this:

```
LISTBLD inputfile outputfile ONLY
  or
LISTBLD inputfile outputfile ONLY ADD
```

If `inputfile` is of the form `@filelist`, LISTBLD with the `ONLY` and `ADD` options can be used to merge two file lists together, eliminating duplicates.

Similarly, you can remove names from a list with the `REMOVE` option, like this:

```
LISTBLD inputfile outputfile REMOVE
```

By using wild-cards and file lists for `inputfile`, you can easily build file lists representing complex relationships between files. For instance, the following example builds a file list named `Z.LIS` containing all the assembler source files in the current directory plus all the source files for `A.EXE` and `X.EXE`, except for the standard include file `STDIO.H` and those include files used by `B.EXE`:

```
LISTBLD A.C Z.LIS NEST
LISTBLD X.C Z.LIS NEST ADD
LISTBLD *.ASM Z.LIS ONLY ADD
LISTBLD B.C TMP.LIS NEST
LISTBLD @TMP.LIS Z.LIS REMOVE
LISTBLD STDIO.H Z.LIS REMOVE
DEL TMP.LIS
```

The resulting file list can be used to specify input files for most programs in the TLIB package. For instance, to convert blanks to tabs in the files listed in `Z.LIS`, you could do:

```
MD TEMP
TABS IN @Z.LIS TEMP\*.*
COPY TEMP\*.*
ECHO Y | DEL TEMP\*.*
RMDIR TEMP
```

By default, LISTBLD records only file names. However, if you specify the `RELATIVEPATHS` option, it can also store "relative" paths along with the file names. The relative paths are relative-to the current subdirectory.

When the `REMOVE` option is specified without `RELATIVEPATHS`, then any relative paths are ignored, and LISTBLD removes every entry in the output file list which matches the specified input file(s).

When the `REMOVE` option is specified with `RELATIVEPATHS`, the complete path+name must match for each entry to be removed.

For example, suppose the `EXCLUDE.LIS` and `MYFILES.LIS` file lists are as follows:

```
exclude.lis       myfiles.lis

iostuff.c         aa\iostuff.c
f\analyze.h       iostuff.c
                  bb\iostuff.c
                  analyze.h
                  aa\analyze.h
                  compute.c
                  f\analyze.h
```

Then specifying `REMOVE` without `RELATIVEPATHS`...

```
LISTBLD @exclude.lis myfiles.lis REMOVE
```

results in:

```
myfiles.lis

compute.c
```

and several warnings about duplicate entries being deleted (because the multiple “`iostuff.c`” and “`analyze.h`” entries are not allowed when `RELATIVEPATHS` isn't specified).

However, specifying `REMOVE` with `RELATIVEPATHS`...

```
  LISTBLD @exclude.lis myfiles.lis REMOVE RELATIVEPATHS
```

results in:

```
myfiles.lis

aa\iostuff.c
bb\iostuff.c
analyze.h
aa\analyze.h
compute.c
```

LISTBLD can also create file lists containing version number information. To do this, use the `ONLY` option and put the version number specification on the LISTBLD command line, like this:

```
 LISTBLD A.C Z.LIS ADD ONLY 5.*
```

This would add or change the file list entry for `A.C` to include the version specification “`5.*`”. Then when you use the `Z.LIS` file list to specify a set of files to TLIB's E, U, or S command, you would be selecting the latest branch version (5.1, 5.2, 5.3, etc.) instead of the latest trunk version.

LISTBLD also supports the `DOTSTAR` option for converting file lists with exact version numbers (or snapshot `.bat` files) into “floating” version label file lists (in which, for each file name in the list, the associated version number ends in “*”). However, this feature is not often used anymore, since TLIB 5.50's named project levels provide a similar facility much more conveniently.

For more information on using version numbers in file lists, see p. 209.

Example :

The “`REMOVE`” option, to specify files to be removed from a file list:

old list `XYZ.LIS`:

```
   MAINFILE.C
   STDIO.H
   IOPKG.C
   LOWLEVEL.ASM
   MAKEFILE
```

command:

```
  LISTBLD *.C XYZ.LIS REMOVE
```

new list `XYZ.LIS`:

```
   STDIO.H
   LOWLEVEL.ASM
   MAKEFILE
```

Note that wildcards and file lists can be used with any of the LISTBLD options. This provides a very flexible mechanism for manipulating file lists.

Example :

To combine two file lists into a single, larger file list (with duplicates removed):

```
  LISTBLD @NEWNAMES.LIS MAINLIST.LIS ADD ONLY
```

The lists `NEWNAMES.LIS` and `MAINLIST.LIS` will be combined; the composite list is `MAINLIST.LIS`.

Example :

The `REMOVE` option can be used to "subtract" two file lists. If you wanted to remove from `MAINLIST.LIS` all the names in `JUNKLIST.LIS`, you could do this:

```
  LISTBLD @JUNKLIST.LIS MAINLIST.LIS REMOVE
```

Example :

The following command would build a list of all the "`.C`" files in the current directory:

```
  LISTBLD *.C BIGLIST.LIS ONLY
```

Example :

The following command would build a single, composite list of all the "`.C`" files in the current directory, plus all the include files used by the `.C` files (it scans for include files because `ONLY` isn't specified):

```
  LISTBLD *.C BIGLIST.LIS
```

Example :

The following would build a file list, called `MISSINCL.LIS`, listing all the include files which are *not* in the current directory, but which are used by Pascal programs in the current directory:

```
  rem  1st, build list of .PAS files & files they "include"
  LISTBLD *.PAS MISSINCL.LIS
  rem  Then build list of all files in the current directory
  LISTBLD *.* TEMP.LIS ONLY
  rem  Then "subtract" the second list from the first
  LISTBLD @TEMP.LIS MISSINCL.LIS REMOVE
```

If you forget how to use LISTBLD, run it with no parameters, and it will display a short "help" message.

File Lists and Snapshots

The snapshot files (version labels) that are created by TLIB's S command are just normal file lists, except that they have version numbers specified for each file name. That is, each line of the snapshot file begins with a source file name, which is followed by the version number specification. For instance, a snapshot recording the current versions for `a.c` and `b.c` might look like this:

```
A.C v=5
B.C v=7
```

This example records the fact that version 5 is the current version for `a.c` and version 7 is the current version for `b.c`. Note that TLIB also accepts an older format which omits the “`v=`”. For example:

```
A.C 5
B.C 7
```

Because file lists (both regular file lists and snapshot files) are so simple, it is easy to create or modify them manually, with a text editor. You could, for instance, for some special purpose, have a snapshot version label file which, for `a.c`, specified the latest version within a particular branch, rather than a specific version number:

```
A.C 5.*
```

Version specifications like that, which end in an asterisk, are called *floating* version numbers. Version numbers which do not contain asterisks are called *fixed* (or *specific*) version numbers. Although TLIB's S (snapshot) command creates snapshots which contain only fixed version numbers, there is nothing to prevent you from creating and using floating version numbers in a manually created or modified snapshot file.

For instance, the following file list specifies the latest trunk version of module `A.C`, plus the latest descendent of branch version 5.1 of module `B.C` (i.e., version 5.*), and version 12.2 of module `D.C`:

```
A.C *
B.C 5.*
D.C 12.2
```

If there are version numbers with each file name in the file list, then it is a *version label file list* or *snaphot*, because it “labels” the versions of each of a set of source files. If the version numbers end in asterisks, it is a *floating* version label file list, because the effective version numbers “float” to the highest version numbers on the specified branches. If the version numbers do *not* end in asterisks, it is a *fixed* version label file list, because the versions are “fixed” at the specified values.

Note that if you use a file list to specify files to TLIB, then any version specifications within the file list will be respected by TLIB unless you specifically force TLIB to use a particular version.

Thus, if you used the above file list (called `3files.lis`) to specify files to TLIB's E (extract) command, you'd get the latest trunk version of `A.C`, but you'd get the latest branch from version 5 for `B.C`, and version 12.2 of `D.C`:

```
tlib e @3files.lis
```

However, if you wanted to extract the latest trunk version of all three files, you could do so by using the “S” suffix to specify the desired version to TLIB, overriding the version numbers in `3files.lis`, like this:

```
tlib es @3files.lis *
```

Note to users of TLIB 3.x and 4.x: Earlier versions of TLIB did not support the S (snapshot) command. Instead, TLIB came with a program called TLIBSNAP, which produced snapshot files that resembled `.bat` files containing TLIB commands. Although TLIBSNAP is obsolete, TLIB can still recognize and use TLIBSNAP's old `.bat` format snapshot files. You use them exactly as you would use the snapshot files produced by TLIB's S command. For example:

```
tlib e @oldsnap.bat
```

As an example of how LISTBLD can be used to create file lists for the S (snapshot) command, here is a little `.bat` file which you could use to take a snapshot of an entire software package (consisting of four programs) written in Pascal, whenever you release a new version:

```
REM snapshot SnazzyWrite package into snazzy.snp,
REM then add snazzy.snp to its TLIB library.
listbld snazwrit.pas snazzy.lis
listbld snazspel.pas snazzy.lis add
listbld snazthes.pas snazzy.lis add
listbld snazinst.pas snazzy.lis add
tlib s snazzy.snp @snazzy.lis
tlib c locking_n u snazzy.snp
del snazzy.lis
del snazzy.snp
```

Keywords

TLIB can insert any of several kinds of information into your source file when you extract it from a TLIB library file (text format only; this feature is not supported for `FileType binary`). The information TLIB inserts is determined by your choice of the "keywords" which you use in "keyword format strings" in your source files. Currently supported keywords are:

`%v` **V**ersion number.

`%f` **F**ile date & time for this version (or date alone, if "`LOGTIME N`" is configured).

`%d` File **d**ate for this version.

`%t` File **t**ime for this version (do not use if "`LOGTIME N`" is configured).

`%w` **W**ho last checked it in (for multi-user environments, if "`LOGUSER Y`" is configured). This is *not* necessarily the *current* user ID.

`%l` **L**ock status. This is the user ID of the programmer who checked this file out for modification, or else the special string `***_NOBODY_***` if the file was extracted in "browse mode" (with EB). (Note: "`l`" is a lower-case `L`, not the number one.)

`%n` File **n**ame.

`%%` Literal percent sign "**%**"

Note: TLIB can also insert a complete revision history comment block into your source code. See Revision History Logging, p. 222.

Keyword information is inserted in the source code in such a way that it can be automatically removed when the modified module is added to the library file (with the U or N command).

To use TLIB's keywords, you must do several things.

The **first step** is to add the **KeyFlag** configuration parameter to your TLIB configuration file (usually named `TLIB.CFG`). Also, examine the configuration file to make sure that you are not using `FILETYPE BINARY` for the files that you want to have keywords in.

(The second and third steps are to add *two* additional lines to each of your source files, as described below.)

The *keyword flag line* is a line in your source file containing the *keyword flag*, a special string of characters starting at a specified column number.

The `KEYFLAG` configuration parameter is what you use to tell TLIB what keyword flag it should look for in your source files. The `KEYFLAG` parameter specifies a column number and a quoted string, separated by a comma. (*Note:* there must *not* be a blank next to the comma.) For instance, if `KEYFLAG` was configured in `tlib.cfg` as:

```
keyflag 1,"; ***keyword-flag***"
```

then a flush-left MASM-style comment line in your source file could be the keyword flag line, like this:

```
; ***keyword-flag*** "version %v"
```

More flexibly, you could configure:

```
keyflag 3,"***keyword-flag***"
```

This specifies that the keyword flag indicator in your source file starts in column 3 and consists of the 18 characters, “`***keyword-flag***`”. This `KEYFLAG` would match any of the following example keyword flags embedded in your source code file:

```
{ ***keyword-flag*** '%v %w %f' }     (Pascal)
/****keyword-flag*** "version=%v" */  (C)
; ***keyword-flag*** '%f %10w'        (MASM)
```

Note: the first column is column one, not zero. Configuring “`keyflag 1,`...” indicates that the keyword flag indicator is “flush-left” (not indented at all) in your source file. Configuring “`keyflag 3,`...” means that in your source code the keyword flag has *two* (not three) blanks (or other characters) preceding the configured string.

The **second step** for using keywords is to add the keyword flag to your source file(s). It should generally be part of a comment line, and it should be immediately followed by a quoted "keyword format template." The keyword format template tells TLIB what to insert into the next line between the quote marks. The quote (delimiter) character must be the first non-blank character after the keyword flag. For example, in the sample Pascal keyword flag line, above, the keyword flag is "`***keyword-flag***`", the delimiter is an apostrophe (`'`), and the keyword format template is `'%v %w %f'`.

Note that the keywords must be lower-case (`%v`, `%w`, `%f`, etc.); you cannot use upper-case keywords (`%V`, `%W`, `%F`, etc.).

The **third step** is to add to your source file(s), on the line immediately following the keyword flag line, the statement in which you want the keyword information inserted. The line must include exactly two delimiters (quote marks). The keyword information will be substituted between the delimiters according to the format template which you specified on the previous line (the keyword flag line).

For the following examples, assume that this `KEYFLAG` definition has been configured:

```
keyflag 3,"***keyword-flag***"
```

Example #1 (MASM):

If version 14 in the library file contains the following:

```
  DB 'version '
; ***keyword-flag*** '%v'
  DB ″,10,13
```

...then it will be extracted as:

```
  DB 'version '
; ***keyword-flag*** '%v'
  DB '14',10,13
```

Example #2 (C language):

If version 5.2.1 in the library file is dated 9-Jul-87,11:23:00 and contains:

```
/****keyword-flag***  " %v %f" */
char version[ ] = "";
```

...then when it is extracted it will contain:

```
/****keyword-flag***  " %v %f" */
char version[ ] = " 5.2.1 9-Jul-87,11:23:00";
```

Example #3 (Pascal):

If version 5.2.1 in the library file is dated 9-Jul-87 (with no time recorded because `LogTime N` is configured) and it contains:

```
{ ***keyword-flag***  'version %v, date=%f' }
const version = ”;
```

...then when it is extracted it will contain:

```
{ ***keyword-flag***  'version %v, date=%f' }
const version = 'version 5.2.1, date=9-Jul-87';
```

Example #4 (Pascal):

If version 5.2.1 in the library file contains:

```
(****keyword-flag***  #%v# *)
(*This is a comment.  This is version ##, within
the comment.*)
```

...then when it is extracted it will contain:

```
(****keyword-flag***  #%v# *)
(*This is a comment.  This is version #5.2.1#, within
the comment.*)
```

By inserting an integer “field width” after the “`%`” of a keyword, you can force TLIB to blank-pad the corresponding value to the desired size. This can be useful when you need to insert the information in a fixed-width field, as in the following Turbo Pascal example:

Example #5 (Turbo Pascal):

If version 5.2.1 in the library file contains:

```
type pstr18 = packed array [1..18] of char;
{ ***keyword-flag***  'version %10v' }
const version: pstr18 = '                  ';
```

...then when it is extracted it will contain:

```
type pstr18 = packed array [1..18] of char;
{ ***keyword-flag***  'version %10v' }
const version: pstr18 = 'version 5.2.1     ';
```

Another warning: the column number specified for a `KEYFLAG` does *not* properly count columns in lines containing tab characters. Also, TLIB does not properly handle leading and trailing blanks within a `KEYFLAG` string, nor does it transparently match tabs and blanks.

Therefore, we recommend that you:

1) specify a column in the range 1-8 *and*

2) don't use tabs or multiple blanks in the `KEYFLAG` string

The use of the first three keywords is fairly obvious, but the `%l` keyword needs a bit of explanation. This keyword is only useful when locking is enabled. It is designed to warn you if you are about to make changes to a file which you do not have checked-out for modification. To use it, you'll also need to set either the “`DELETESRC Y`” or the “`FIXKEYWD Y` configuration parameter, so that checked-in files are not left around which appear to be still checked out.

The `%l` keyword works well because most programmers' editors start out by displaying the first 24 lines or so of the text file when you begin an edit session. So, a prominent warning inserted at the top of the file will be seen whenever anyone edits the file.

To use this keyword, insert a `KEYFLAG` comment at the beginning of each text file, similar to the following C language example:

```
#if 0
/* --=>keyflag<=--  " checked-out to %l " */
/*     " "  */
```

```
#endif
```

When “Dave” checks out the file for modification with the U or X command, the beginning of the file will look like this:

```
#if 0
/* --=>keyflag<=--  " checked-out to %l " */
/*     " checked-out to DAVE " */
#endif
```

But if the file was extracted with the EB command (browse mode), the beginning of the file will look like this:

```
#if 0
/* --=>keyflag<=--  " checked-out to %l " */
/*     " checked-out to ***_NOBODY_*** "  */
#endif
```

Note that the `%l` keyword cannot do the “right” thing if you use the O and I commands to manage your check-in/check-out locking. What is inserted depends solely upon the type of extract command which you use. The EB command is for “browse mode”. The E command checks out the file for modification.

We're grateful to Mr. Larry Young for the idea behind this feature.

For an often-superior alternative to the `%l` keyword, see the `READONLYB` configuration parameter, p. 280.

Two notes:

1) If you use TLIB's “Revision History Logging” feature (p. 222), then you'll notice that the `KEYFLAG` parameter is similar to the `LOGFLAG` parameter. If you configure both a `KEYFLAG` and a LogFlag, TLIB *requires* that the starting column numbers and the first two characters of the flag strings for the `KEYFLAG` and `LOGFLAG` be identical (this requirement is imposed for performance reasons).

2) Keywords and Revision History Logging are only supported for text format library files, not for binary files. If keywords do not seem to be working, check that you have not configured `FileType binary`. Also, note that the `FileType` configuration parameter only affects the creation of new library files; if a library file already exists, then changing the

`FileType` parameter will have no effect on it. You can easily tell whether a particular TLIB library file is text format or binary format by inspecting the first three bytes in the file. If the file begins with "`.Vc`" then it is in binary format; if it begins with "`.V `", "`.V_`", or "`.Vt`" then it is in text format.

If you need to convert one or more TLIB libraries from `FileType Binary` to `FileType Text`, or vice-versa, you can use the TLIB-to-TLIB conversion utility, `TLIBTLIB.EXE` (or `TLIBTLIB.AWK`). (With some versions of TLIB, these conversion utilites may be found in the `CONVERT1.ZIP` or `CONVERT2.ZIP` archive.) See the `*.TXT` or `*.DOC` files in that archive for instructions.

Keyword-based version number checking

TLIB 5.0 added "keyword-based version number checking," which produces a warning if you update a library with a new version of a module and the new version contains an old `%v` keyword which is not the predecessor of the new version. If you put a `KEYFLAG` with "`%v`" in each of your source modules, then this new feature will provide an additional measure of safety to help programmers notice when they are about to "undo" someone else's changes.

Caveat: TLIB must be able to find the version number in the formatted keyword information. More specifically, you must not have something adjacent to the `%v` which could be confused with a version number, and if the `%v` is not the first keyword in the keyword format string then it must be preceded by something distinctive.

Usually this is not a problem, but if you try, you can find a way to confuse TLIB (and perhaps yourself, too). For instance, suppose that you configured:

```
  keyflag  3,"-=>keyflag<=--"
```

...and suppose your `KEYFLAG` format looked like this:

```
 /*-=>keyflag<=-- "version=%v0" */
 char embedded_id[42] = "version=960";
```

`...then TLIB will think you have version 960 instead of version 96.`

Less obviously, you can confuse TLIB by having another keyword before the `%v`, and not have something distinctive between them. For instance:

```
/*-=>keyflag<=-- " %n %v " */
char embedded_id[42] = " MYFILE.C 96 ";
```

...will confuse TLIB because the single blank before the `%v` is insufficiently distinctive (since there are other single blanks elsewhere in the keyword format string). However, the following will work okay:

```
/*-=>keyflag<=-- " %n  %v " */
char embedded_id[42] = " MYFILE.C  96 ";
```

...because the double blank before the `%v` is unique in the keyword format string.

Keyword-based version number checking is especially useful for those who have locking disabled, perhaps because the programmers' workstations are not connected via a network ("sneakernet"), the lack of which makes check-in/out locking less convenient). However, it provides an extra measure of safety regardless of your development environment.

Other changes in TLIB 5.0

TLIB 5.0 and later let you specify "`%%`" in a keyword format string to generate "`%`" in the formatted keyword information.

Also, TLIB now displays a warning if the `%w` keyword is used but `LOGUSER N` is configured.

What if it doesn't work?

If keywords don't seem to be working for you, then check the following:

o Make sure that the configured `KEYFLAG` string is *identical* to the keyword strings in your source files. Beware of case mismatches and tabs.

o Count the column numbers carefully. If you've configured "1" as the starting column, then the keyword string must start in the leftmost column of your source file.

`o` Make sure there is only one `KEYFLAG` parameter configured. If there is more than one, then all but the last is ignored (except that you can use `IF`/`ENDIF` blocks to specify different keyword flags for different files). If you use `INCLUDE` directives in your main TLIB configuration file, be sure to check the included configuration files, too.

`o` Make sure that TLIB is reading the right configuration file. How can you tell? Add a line of gibberish to the configuration file. If TLIB doesn't complain, then it isn't reading that file (see p. 250). Also, you can obtain a dump of TLIB's actual configuration settings by using the "`-c`*tempfile*" command-line option, as in "`tlib -ctemp.tmp q`".

`o` If the file is more than a few thousand lines long, make sure that the keywords are near the beginning. For multipass libraries, TLIB can only process keywords that appear within the first `PASSSIZE` (`MAXLINES`) lines (default 4000). If you need to use keywords near the end of a very large file, you can create the TLIB library with `PASSSIZE` set as high as 16000, and use `TLIBX.EXE` (the DOS-extended version of TLIB). That will allow you to use keywords anywhere within the first 15999 lines of the file.

`o` Make sure that your TLIB library was not created with `FILETYPE BINARY` configured. Examine the TLIB library file with a text editor. The first two characters are "`.V`". If the library is in text format, then the third character is "`_`", "`t`", or a blank. If the third character is "`c`" or "`d`" then the library is in binary format, and you'll have to create a new libary file (with `FILETYPE TEXT` configured) to make keywords work, or convert the library to text format format with `TLIBTLIB.EXE` or `TLIBTLIB.AWK` (p. 296).

Tlibscan

TLIBSCAN is a fast file-scanning program, similar to fgrep. It can search any kind of file, including binary files, and display ASCII data found in the files. TLIBSCAN is especially useful for identifying .EXE files which have embedded keyword information (like WHAT or IDENT from other systems), but it can also be used as a general-purpose file searching tool.

If you use the TLIB keyword facility to embed version-specific information in your source files, TLIBSCAN can find it in the .EXE file. By default, TLIBSCAN will search for the string, “`&(#)`”, and it will display text found after the “`&(#)`” and up to the first control character (such as a 0-byte or carriage return). Seven command line options are available to change its behavior. For a list of options, run TLIBSCAN with no parameters.

Here are examples of how to use keywords to embed scannable information in your programs. Both assume that you've configured TLIB similar to this:

```
keyflag 3,"-=>keyflag<=--"
```

Example #1 (C):

```
/*-=>keyflag<=-- "&(#)MAKE #%v %f" */
static char embedded_id[] = "&(#)MAKE #63 2-May-99,9:23";
```

Example #2 (Turbo Pascal):

```
{--=>keyflag<=-- '&(#)%n #%v %f'}
embedded_id: string[41] = '&(#)TST.PAS #8 2-May-99,9:30'#0;
```

We have embedded scannable information in several of the TLIB programs. So, to try out TLIBSCAN on the TLIB programs, enter:

```
TLIBSCAN *.EXE
```

Revision History Logging

Revision history logging is a feature which allows automatic insertion in your source file of a revision history comment block (text files only; this feature is not supported for `FileType binary`). The revision history is similar to the output of TLIB's L (list versions) command. *If you do not need this feature, you need not read this chapter.*

Several TLIB configuration parameters are used to tell TLIB how to automatically insert a “revision history” of version comment lines into your source code when you extract the source file from the library file. The configuration parameters allow you to specify the format and placement of the “revision history log” in your source file (typically in a “comment block”).

Three parameters (**logFlag**, **logPrefix** & **logSuffix**) are used to support the automatic insertion of a “log” of version definition comment lines into the source file. The `logFlag` parameter is used to determine where in the source file the log will be inserted. The `logPrefix` and `logSuffix` are optional parameters which are used to format the log information when it is inserted in the source file.

The log must be both preceded and followed by a “LogFlag,” which is very similar to a `KEYFLAG` (for keywords, p. 212).

If you configure both a `KEYFLAG` and a `LOGFLAG`, then TLIB requires that the starting column numbers and the first two characters of the flag strings for the `KEYFLAG` and `LOGFLAG` be identical (this requirement is imposed for performance reasons).

The `LOGFLAG` is a line containing a special string of characters starting at a specified column number, so the configuration parameter for a `LOGFLAG` consists of a column number plus a quoted string, separated by a comma. (*Note:* there should *not* be a blank next to the comma.) For instance, if the `LOGFLAG` was configured as:

```
 logflag 1,"C ***revision-history***"
```

then a pair of Fortran-style comment lines would delimit the log. Or, more flexibly, you could specify:

```
logflag 3,"***revision-history***"
```

which would match, for instance, these Fortran, Pascal, C & MASM comments:

```
C ***revision-history***    (Fortran)
{ ***revision-history*** }  (Pascal)
/****revision-history****/  (PL/I or C)
; ***revision-history***    (MASM)
 (***revision-history***    (Pascal: begin-log)
  ***revision-history***)   (Pascal: end-log)
```

This `LOGFLAG` is compatible with the example `KEYFLAG` described earlier, since the `KEYFLAG` and `LOGFLAG` both start in column 3 and begin with "**".

Note that when you create the original source file, you must include *two* `LOGFLAG` lines, one after the other, in your source file. Later, when you extract the source file, the log will be inserted between them. Be certain that you have *two* logflags, since everything after the first one and before the second will be deleted when you use the U or N command to put the source file into the library file!

For C programmers, we recommend that the log be placed in an `#if/#endif` block, like this:

```
#if 0
  ***revision-history***
   . . .
  ***revision-history***
#endif
```

Note that the TLIB U (update) and N (new library) commands will not modify the source file to update your revision history log if you've configured `FIXKEYWDS N`, so to get an up-to-date version log inserted in your source file after an update, you may have to re-extract the source file (or just configure `FIXKEYWDS Y`).

Another warning: the column number specified for a `LOGFLAG` does *not* properly count columns in lines containing tab characters. Also, TLIB does not properly handle leading and trailing blanks within a `LOGFLAG` string, nor does it transparently match tabs and blanks.

Therefore, as with KeyFlags, we recommend that you:

1) specify a column in the range 1-8 *and*

2) don't use tabs or multiple blanks in the `LOGFLAG` string

LogPrefix and **LogSuffix** specify what, if anything, should precede the version comment lines when they're inserted in the source code as a revision history log. Both the `LogPrefix` and `LogSuffix` parameters specify a column number (where the prefix or suffix is to be inserted) and a quoted string. The version comment lines will be truncated as needed to fit between the prefix and suffix, so you may wish to restrict the length of a version comment line by using the `LogWidth` parameter (p. 267).

Restriction: the `LogPrefix` can't extend past column 80.

Here's a `logflag`, `logprefix` and `logsuffix` for use with Pascal programs:

```
 logflag 1,"{--=>revision history<=--}"
 logprefix 1,"{"
 logsuffix 79,"}"
```

Note that with this choice of `LOGFLAG` and `LOGPREFIX`, it is impossible for a version comment line to accidentally match the terminating logflag, since the `LOGFLAG` always has “`-`” in column 2, but the version comment lines will have either a blank or a digit in column 2 (try it and you'll see why).

One final warning about `LOGFLAG`s: if you use them, you may have to be careful not to put any “close comment” delimiters into your version definition comment lines, lest your compiler try to compile part of the version log! This is obviously not a problem in languages which let you denote a comment by preceding it with a special character (e.g., “`;`” for MASM). For C programs, the problem can be avoided by putting the `LOGFLAG` pair within an “`#if 0`”/“`#endif`” block. But Pascal programmers, watch out!

Note: if you program multiple languages, you may need to configure several styles of revision history comment block. You can use TLIB's `IF` and `ENDIF` configuration parameters for this; see p. 321.

Cmpr delta generator

CMPR is a text file compare utility based on TLIB. Its output is a *delta:* a description of the differences between a “new” file and an “old” file. The delta can be used to reconstruct the “new” file from the “old” file.

The primary purpose of this utility is to allow transfer of source code updates in a compact format over low speed telecommunication lines.

Note #1: for a more “human-friendly” (informative) compare utility, try `COMPARE.EXE` (from `PUBLIC.ZIP`). Or, for the nicest compare utility we've seen for ASCII text, buy *DELTA*™ (list price $100) from OPENetwork, 215 Berkeley Place, Brooklyn, NY 11217. Tel: (718) 398-3838.

Note #2: For comparing specified versions of a file in a TLIB library, or for comparing an already-extracted file to another version which is in a TLIB library, you can use the included `COMPAR.BAT` script. It simply invokes TLIB to extract the desired version(s) into temporary file(s), uses `COMPARE.EXE` to compare them, and then uses `MORE` to view the differences. You can easily modify it to use a different compare program (such as *Delta*), and/or to use a different tool to view the results, if you wish.

Modes

The output is in any of several formats, selected according to a “mode” parameter. The default mode (“`NORMAL`”) is a TLIB-style differences description. If you have the old version of a file, plus the “delta file” produced by CMPR, then you can reconstruct the new version by using a simple `.bat` file which builds a temporary TLIB library file, appends the delta to it, and then extracts the new version.

See the `UNCMPR.BAT` file for an example of how to do this.

You can also select a CMPR mode of `ONEPASS`, which is just like the normal mode except that it is guaranteed that the lines referenced in `.C` (copy) edit commands will be monotonically increasing; this may simplify your task if you wish to translate the delta file into another format, for use with a “batch editor.”

The third supported mode is `INSDEL`, which is similar to onepass mode except that the edit commands consist of inserts and deletes instead of inserts and copies. This format is described below. We've also included a simple Pascal program, DSAPP ("Delta Script Applier"), which can apply insdel format deltas to a file. DSAPP is supplied as both source code and `.EXE` file; see p. 230.

The fourth supported mode is `SPERRY`, which causes CMPR to output its delta file in a format which is compatible with the Sperry Unisys SDF/SIR$ program. Thus, if you keep source files on both a PC and a Sperry Unisys mainframe computer, you can edit the PC version and upload just the changes (to make the file transfer faster). You can then easily reconstruct the new version on the mainframe by using SDF/SIR$.

The fifth supported mode is `PANVALET`, which is similar to `SPERRY` mode except that the output is compatible with Pansophic's PANVALET program management system. The format uses the `++C` and `++D` subcommands of the `++UPDATE` command, as described in the PANVALET User Reference Manual. The special PAN#1 column one "`$`" input character combinations are translated properly (e.g., "`$*`" is substituted for "`/*`"). We've also included a PANVALET/P-COMPARE format delta script applier, `DSAPPPAN.EXE`.

"PANVALET" is a registered trademark of Pansophic Systems, Inc.. The PANVALET User Reference Manual, number OSUP11.0-8412, is available from Pansophic Systems, Inc., Lisle, IL.

The sixth supported mode is `LIBRARIAN`, which generates a delta script file that is compatible with Applied Data Research's LIBRARIAN program management system. This format uses the "`-INS`" and "`-DEL`" control statements as described in the ADR LIBRARIAN User Reference Manual. JCL, POWER and LIBRARIAN control statements are automatically translated with the appropriate LIBRARIAN "conversion characters" (e.g., "`/:`" is substituted for "`/*`"). We've also included a LIBRARIAN format delta script applier, `DSAPPADR.EXE`.

"LIBRARIAN" is a registered trademark of Applied Data Research, Inc.. The ADR LIBRARIAN User Reference Manual, number SV2G-10-30, is available from Applied Data Research, Route 206 and Orchard Rd., CN-8, Princeton, NJ 08540.

The seventh supported mode is `IBMUPDATE`, which generates a delta script file that is compatible with IBM's UPDATE mainframe utility. This format utilizes "`./ I`" and "`./ D`" control statements. A sample VM/CMS EXEC for applying deltas is included with TLIB, in the file named `IBMUPDAT.EX`.

Add the "`SMOOTH`" option to make CMPR generate more concise deltas for large (multi-pass) files in the `ONEPASS`, `INSDEL`, `SPERRY`, `IBMUPDATE`, `PANVALET`, and `LIBRARIAN` output formats.

The `SMOOTH` option tells CMPR to "resynchronize" between passes, so that the output does not show a bogus edit for lines that have moved across the `DIFFLINES`-long line buffer boundary. This effectively makes `DIFFLINES` vary from pass to pass, according to the contents of the files, for files that are longer than the configured `DIFFLINES` number of lines. The default `DIFFLINES` size is 3000, so the `SMOOTH` option does not affect operation with files that are smaller than that.

If you are using CMPR to generate TLIB-compatible deltas ("`NORMAL`" mode), then you should not specify the `SMOOTH` option, since TLIB uses only fixed-length pass sizes, and cannot handle the variable-length pass sizes which result from the `SMOOTH` option.

Three other CMPR options are provided for use in combination with mainframe delta formats:

`WIDTH=`*nn* Truncates lines at column *nn*.

`IGNORE=`*nn* Ignore first *nn* characters on each line.

`SEQ10` Sequence by 10 instead of 1.

Use the `WIDTH` option with any CMPR output format to remove sequence number information from the input files. For instance, to remove sequence numbers from columns 73-80, specify `WIDTH=72`.

Similarly, you can use the `IGNORE` option to cause CMPR to pretend that the first *nn* characters on each input line are blank. This is useful for COBOL programs with sequence numbers at the beginning of each line.

CMPR does not recognize sequence number information in input files. The line numbers which it uses simply refer to the position in the file. That is, line N normally refers to the N-th line in the file.

However, in some mainframe environments lines are customarily referred to by sequence numbers which increment by ten. That is, the first line is number 10, the second is number 20, etc..

To accommodate mainframe programs which expect this kind of sequence number, CMPR can optionally multiply all line numbers by ten when generating mainframe delta script files. Specify the `SEQ10` option if you require sequence numbers which increment by 10.

Since CMPR cannot recognize sequence numbers in columns 73-80, you may need to do a resequence operation on your mainframe file before applying the delta script.

If you have a requirement for a special CMPR output mode, please contact Burton Systems Software. We will be happy to work with you to try to meet your needs.

Output Format Detail

NORMAL and ONEPASS Mode:

In normal and onepass modes, CMPR outputs a TLIB-style differences description, consisting of a `.V` header line followed by `.I` (insert) and `.C` (copy) commands.

The `.I` edit command format is...

```
.I N
newline1
newline2
  ...
newlineN
```

...where `N` is the number of new lines to be inserted.
The `.C` edit command format is...

```
.C X Y
```

...where `X` is a line numbers in the ‘old’ input file. `Y` is optional. If it is absent, a single line is to be copied. If `Y >= X` then `X` and `Y` represent the first and last old line numbers in a range of lines. If `Y < X` then `Y` is one less than the number of lines to be copied.

You could write a program to combine the delta file with the ‘old’ file to reconstruct the ‘new’ file like this:

1) Start out with an empty ‘new’ file buffer

2) Examine next edit command (`.I`=insert or `.C`=copy)

3) Append inserted or copied lines to ‘new’ file buffer

4) Repeat 2 & 3 until there are no more edit commands

In fact, this is exactly what TLIB does when it is reading a library file.

INSDEL Mode:

In insdel mode, the CMPR output format consists of inserts and deletes instead of inserts and copies. The difference between insert/copy format and insert/delete format is in how you go about combining the delta file with the ‘old’ file to reconstruct the ‘new’ file:

1) Start out with a copy of the ‘old’ file

2) Examine next edit command (`.I`=insert or `.D`=delete)

3) If it is an insert, add the lines at specified position

4) If it is a delete, remove the specified lines

5) Repeat 2-4 until there are no more edit commands

In insdel mode, the `.I` (insert) edit command format is...

```
.I W N
newline1
newline2
  ...
newlineN
```

...where `N` is the number of lines to insert, and `W` is where to insert them. More precisely, `W` is the line number in the ‘old’ file after which the new lines should be inserted (to insert before the first line, use “`.I 0 N`”).

The `.D` (delete) edit command format is...

```
 .D W N
```

...where `N` is the number of lines to be deleted, and `W` is the (old) line number of the first line to be deleted (`W` is always greater than 0).

We have also included a small program called **DSAPP** with TLIB. DSAPP (“delta-script application”) can be used to apply a CMPR-generated “delta” file (in insdel format) to an “old” file, creating a “new” file. DSAPP is about 100 lines of Turbo Pascal; both source and `.EXE` are provided.

Because it is uncopyrighted, you may give copies of DSAPP to anyone you wish. You can use it to speed distribution of source code or documentation updates via slow telecommunications links. You would generate an insdel format delta from the “old” and “new” files using CMPR. Send the (small) “delta” file to someone who already has the “old” file, and they can regenerate the (large) “new” file using DSAPP.

DSAPP is similar in function to `UNCMPR.BAT`. But if you use DSAPP, the recipient need not own a copy of TLIB. Also, `UNCMPR.BAT` utilizes the default CMPR output format, while DSAPP uses INSDEL format.

We've also included programs similar to DSAPP for the Pansophic PANVALET and ADR LIBRARIAN formats. These programs are called `DSAPPPAN.EXE` and `DSAPPADR.EXE`, respectively. Unlike `DSAPP.EXE`, these programs *are* copyrighted. Run them with no parameters for instructions on how to use them.

Other modes:

When you use CMPR's `SPERRY`, `PANVALET`, `LIBRARIAN` or `IBMUPDATE` modes, the output format is similar in structure to `INSDEL` mode, except for the format of the insert and delete commands. For details, see the appropriate Sperry Unisys, Pansophic Systems, Applied Data Research or IBM documents, respectively.

DOS Errorlevels

CMPR exits with the DOS errorlevel set to indicate whether the comparison failed or succeeded. The errorlevel will be zero if CMPR ran successfully and the two files were not the same.

If the ‘old’ input file does not exist, CMPR will still run, but the “delta” file which results will consist of a single large insert. The DOS errorlevel will be zero. If you need to test for this in a `.BAT` file, use DOS's “`if exist`” construct (see your DOS manual).

If CMPR ran successfully but the two input files were identical, the errorlevel will be 1. CMPR will still create an output (delta) file, but it will be very short (or empty, for insdel and Sperry modes).

If an error occurred (file not found, for instance), then CMPR will return with an errorlevel of 2. The only exception to this is a Ctrl-Break or Ctrl-C from the keyboard, which may cause CMPR to halt with errorlevel 0.

Configuration

CMPR reads the same configuration file as TLIB, but all except four of the configuration parameters are ignored. The four parameters which are not ignored are:

addCtrlZ: Same purpose as for TLIB; determines whether or not a Ctrl-Z character (End-Of-File, code 26) will be added to the output file. Default is N (no). See p. 276.

difflines: Defines the size of CMPR's internal line buffer. This is the number of lines which CMPR will compare at one time. Note that CMPR will handle files which are longer than passsize/maxlines, but the delta may be somewhat less concise. The default is 3000 lines, which is a good choice for most people. If your computer does not have much RAM memory, you may need to reduce difflines.

CmprEntab: Determines whether blanks are compressed to tabs when read by CMPR. This would allow CMPR to match lines correctly regardless of whether they have tabs in them. Note that TLIB and CMPR can only correctly convert standard 8-space tabs. Default is N (no).

CmprDetab: Determines whether tabs are expanded to blanks in the delta file. Default is N (no). Set this to Y if you want tabs expanded in the delta file (see p. 339).

Help Screen

If you run CMPR with no parameters, you will see a help screen similar to the following:

```
Usage: CMPR <oldfile> <newfile> <differencesfile> <options>

<oldfile> and <newfile> are the two input files
<differencesfile> is the output file
<options> are either omitted or:

NORMAL - output is standard tlib “delta” (default)
ONEPASS - same as NORMAL, but .C references are monotonically
increasing
INSDEL   -   similar   to   ONEPASS,   but   translated   to
‘Insert/Delete’ format
SPERRY - output in Sperry Unisys SDF/SIR$ format
IBMUPDATE - output in IBM UPDATE format
PANVALET - output in Pansophic PANVALET format
LIBRARIAN - output in ADR LIBRARIAN format
WIDTH=nn - truncate lines at column nn
SEQ10 - for mainframe formats, sequence line numbers by 10
instead of 1
IGNORE=nn - pretend first nn characters on each line are
blanks

On exit, the DOS errorlevel is set to:
0 - no errors (or aborted with Ctrl-Break)
1 - no errors, but input files are identical
2 - error; <outfile> was not created
```

Tlmerge / Diff3

TLMERGE is a program to merge two sets of revisions, or to effectively "delete" a set of changes from a TLIB library file while retaining later changes. It is used by TLIB's "Migrate" command to migrate/merge changes from one version of your program into another.

DIFF3 is a 16-bit version of TLMERGE.

TLMERGE (or DIFF3) compares three files, a "base" file and two "new" files, and generates a fourth file (the "output") which combines the changes from each of the two "new" files.

Usage

TLMERGE is most commonly used indirectly, by using TLIB's M (migrate) command to generate a migrate2.bat file which contains TLIB and TLMERGE commands to migrate/merge changes from one variant of your program or project into another. Examples would include migrating bug fixes from a currently-release level into a develompent level, or to migrate changes from a standard version into a customized version. If you use TLIB's Migrate command, then you don't need to worry about getting the TLMERGE syntax correct, because TLIB determines what needs to be merged, and generates the proper TLMERGE commands automatically.

However, you can also run TLMERGE (or DIFF3) by itself, from a DOS/Windows command prompt. If a source file has been edited in two different ways, and you need a version which contains both sets of changes, run TLMERGE like this:

```
  TLMERGE basefile file2 file3 outputfile
```

or

```
  TLMERGE basefile file2 file3 outputfile WIDTH=nn
```

where:

`basefile` is the original unchanged file

`file2` is the modified version of basefile with one set of changes

`file3` is the modified version of basefile with the other changes

`outputfile` is created by TLMERGE and contains both sets of changes

`WIDTH=`*nn* truncates lines at column *nn*.

Use the `WIDTH` option to remove sequence number information from the input files. For instance, to remove sequence numbers from columns 73-80, specify `WIDTH=72`.

Alternately, you can think of the parameters like this:

```
  TLMERGE deltaold deltanew inputfile outputfile
```

where:

`deltaold` and `deltanew` are a pair of files which, by the difference between them, describe a “delta” or set of changes

`inputfile` is the file which you need to modify

`outputfile` is `inputfile` modified in the same way that `deltaold` was modified to make `deltanew`

For example, suppose you had a module with two development paths. Versions 1 through 8 are the “main” (trunk) versions. Versions 5.1 through 5.3 are for a special, custom version of the module needed by a favored customer.

If you discovered a bug which was introduced in version 4, then the bug would need to be corrected in both the regular and the customized versions. Rather than manually making the same changes to both current versions, you could make the change to one of them and then let TLMERGE apply the change to the other version. If you made the change to version 8, creating a new, fixed version 9, then TLMERGE can make the same set of changes to version 5.3. Think of version 8 and version 9 as defining the delta, and version 5.3 as the input file. The new, custom module (version 5.4) is the output file:

```
  TLMERGE version8 version9 version5.3 version5.4
```

where:

`version8` and `version9` are the unfixed and fixed trunk versions

`version5.3` is the unfixed customized version

`version5.4` is the fixed customized module, created by TLMERGE

Sometimes TLMERGE might be unable to reconcile the two sets of changes, because, for example, both file2 and file3 contain modifications to the same line(s) in basefile. The changes are said to be conflicting (or have “collided”). TLMERGE will warn you about such conflicts, but you must reconcile them manually.

To help you do this, TLMERGE will insert special “flag” strings in the output file for which you can search with a text editor. By default, the flags look like:

```
 /* ###Change collision detected! filename(s) */
```

If you don't like the default flag, you can change it with the `D3COLLIDE` configuration parameter in your TLIB configuration file (see p. 340).

You can also cause all changed lines to be flagged with an indication of which file they were changed in (see the `D3FLAG2` and `D3FLAG3` parameters, p. 340).

Undo a Revision Without Losing Later Revisions

Occasionally, you may need to undo a set of changes made to one of your source code files without undoing a later series of changes.

For example, suppose that version 4 of `MYMAIL.C` supports 40 byte name records and 5 digit zip codes. In version 5, `MYMAIL.C` was changed to support 100 byte name records. Version 6 was changed to support 9 digit zip codes. Finally, an unrelated bug was found and fixed, creating version 7. The last 4 lines of `MYMAIL.C`'s revision history might look like this:

```
  4 MYMAIL.C 4-Aug-86 5-digit Zip, 40-byte names
  5 MYMAIL.C 20-Oct-86 support 100-byte names
  6 MYMAIL.C 13-Nov-86 use 9-digit Zips
  7 MYMAIL.C 12-Dec-86 Bug fix for files > 64K
```

Now, suppose that you need a new version which is just like version 7, except that it should support 40 byte name records instead of 100 byte name records. In other words, you would like to undo the changes which created version 5 from version 4, but retain the improvements made in versions 6 and 7.

TLMERGE can do this automatically. Simply consider version 5 to be the basefile, and combine the changes needed to convert version 5 back to version 4 with the changes needed to convert version 5 into version 7, like this:

```
 tlib ebs mymail.c4 4
 tlib ebs mymail.c5 5
 tlib e mymail.c7
 diff3 mymail.c5 mymail.c4 mymail.c7 mymail.c
```

In other words, the “delta” is the set of changes needed to convert version 5 into version 4, and the input file is version 7.

Note the use of a handy “trick” in this example: we extracted different versions of `mymail.c` without having to rename them by the simple expedient of chosing alternate file names which still “map” to the same tlib library file (probably `mymail.c$` in this case, if you have configured “`LIBEXT ? $?????`”, as is usual for C programmers).

Configuration

TLMERGE and DIFF3 use the following TLIB configuration parameters:

addCtrlZ: Same as for TLIB & CMPR. (Ctrl-Z for output files? Default is N/no.) See page 276.

difflines: Maximum number of input lines (per file) which will be handled at one time (same as for CMPR). DIFF3 generally works well even if the input files are longer than `DIFFLINES`, as long as `DIFFLINES` is larger than any of the changes. The default is 3000. See page 339.

CmprEntab: Same as for CMPR. (Enable blank-to-tab conversion? Default is N/no.) See page 339.

CmprDetab: Same as for CMPR. (Enable tab-to-blank conversion? Default is N/no.) See page 339.

d3collide: D3collide defines a special line to be inserted in the output file wherever DIFF3 detects conflicting changes in file2 and file3. If this configuration parameter is not defined, a default flag line is used. See page 340.

d3flag2: Because changes to a program may well conflict even if no single line in the base file is altered in both file2 and file3, you may wish to flag all changes, rather than just those which happen to "collide." This parameter is used to describe how to flag all those lines which were modified or inserted in file2. If this parameter is not defined, nothing is inserted. See page 340.

d3flag3: Same as `D3FLAG2`, but for file3 instead of file2.

DOS Errorlevels

TLMERGE and DIFF3 exit with the DOS errorlevel set to indicate whether the comparison failed or succeeded. The errorlevel will be zero if TLMERGE ran successfully and the three input files were not the same.

If the three input files (basefile, file2 and file3) were identical, the DOS errorlevel will be 1. In this case, TLMERGE will still run, but the output file will be the same as the input files.

If an error occurred (e.g., file not found), then TLMERGE will return with an errorlevel of 2. The only exception to this is a Ctrl-Break or Ctrl-C from the keyboard, which causes TLMERGE or DIFF3 to stop with errorlevel 0.

Help Screen

If you run TLMERGE or DIFF3 with no parameters, it will display a short "help" message describing how to use it. The message contains a picture drawn with the IBM PC's extended graphics characters; this picture may not display properly on some computers. However, TLMERGE or DIFF3 will still function correctly on these machines.

Additional features

TLMERGE and DIFF3 have two other minor features, which are used by TLIB's M (migrate) command:

a) TLMERGE and DIFF3 allows an “alias” to be specified for its 2nd and 3rd input files. The alias will be displayed in collision flags instead of the actual file name. TLIB's Migrate command uses this feature to make the collision flags show meaningful version numbers.

The aliases, if present, must be enclosed in `{`curly braces`}`, and must follow the file names immediately, without intervening spaces. For example:

```
  TLMERGE basefil file2{12} file3{9.3} outfil
```

b) TLMERGE and DIFF3 allow specification of a “`line1=`...” parameter, to insert an extra line at the top of the output file. (This is used by TLIB's Migrate command to insert the `CMTFLAG` string.) For example:

```
  TLMERGE basefil file2 file3 outfil line1=[MERGED_12_+_9.3]
```

NS Command – split a library

TLIB was carefully written to run as fast as possible, so that you can make frequent library updates without wasting a lot of time. Eventually, however, if the number of versions in a library file becomes very large (say, over 1000), you may find it convenient to "split" a library file, to improve TLIB's performance or save disk space.

One way to do this would be to simply store the old library file on an archival diskette, and create a new library file (with the N command) from the current version of your source file. However, this would lead to confusion because there would then be two "version ones."

To avoid this problem, TLIB provides the NS (split library) command. The NS command is just like the N (new library) command, except that you must explicitly specify the version number for the first version in the library file; the "S" suffix stands for "specify." (The first version in a library file created by the N command is always number 1.)

You would normally specify a version number which is one larger than the highest version number in the old library file; you can use the L (list) command with the old library file to find out what that number is.

Here is an example showing how to use the NS command:

The library file for `monster.pas` *is using up a lot of disk space, so we decide to split it. Assuming that the TLIB library for* `monster.pas` *is in* `\tlibs\monster.p$s`*, here is how we go about splitting it:*

```
TLIB L MONSTER.PAS
```
(use L command to find that there are 100 versions of `monster.pas`*)*

```
TLIB E MONSTER.PAS
```
(get the latest version from the old library file)

```
REN \TLIBS\MONSTER.P$S *.TL1
```
(save the old library file as `monster.tl1`*)*

```
TLIB NS MONSTER.PAS 101 old lib's v100
```
(start a new library file, with the first version being number 101)

TLIB will now run faster and use less memory, since it doesn't need to read as many old versions each time you do an update or extract.

Retrieving by Date/Time

You can retrieve particular versions by date & time, as well as by version number or by snapshot or file list name. Simply use the ES or EBS command and specify the date and/or time instead of the version number.

If you specify a date/time as the version number for the ES command, TLIB will retrieve the newest version which is not newer than the specified date/time. By default, only trunk versions are examined, but if you are using branching then you can select a particular branch.

For retrieval by date/time to work correctly, the versions in your library file should be in chronological order; that is, the dates of the versions in your TLIB library file should be monotonically increasing (at least within the series of trunk or branch versions which TLIB needs to scan), since TLIB will stop reading versions from the library file when a too-new version is encountered.

If you plan to use this feature then you should *not* configure “`LOGTIME N`”.

You should normally specify the date and time in the format which TLIB uses when displaying the date & time for stored versions. That is, for the date the required format is:

```
 DD-MMM-YYYY
```

where:

`DD` is a 1 or 2-digit day of the month, from 1 to 31
`MMM` is the first 3 letters of the English month name (e.g., Feb)
`YYYY` is the 2 or 4-digit year, 80-99 or 1980-2043

Time of day must be specified in 24-hour format:

```
HH:MM:SS
```

where:

`HH` is the hours, 0 to 23

`MM` is the minutes, 00 to 59
`SS` is the seconds, 00 to 59

If you specify only the time, then today's date is assumed.

If you specify only a date, then 23:59:59 is assumed (so that a version will match if it was created at any time that day). If you specify both date and time, separate them with a comma or slash. TLIB customarily uses a comma, but you can optionally use a slash instead of a comma to separate the date from the time. This is useful if you use the `ES` command with a specified date/time in lieu of a version number, and you want to pass the date/time to TLIB indirectly, as a parameter to a batch file which invokes TLIB.

The problem is that DOS's obnoxious `command.com` converts commas (and semicolons and equal signs) into spaces, so that (for example) a date/time like "`25-Dec-94,12:00:00`" turns into two parameters, "`25-Dec-94`" and "`12:00:00`" when you pass it to a `.bat` file. So, in case you need to do this, TLIB lets you specify the date/time as "`25-Dec-94/12:00:00`" so that `command.com` won't mangle it when you pass it as a parameter to a `.bat` file.

If you are using branching, then you can select by date/time on a particular branch in the library, by specifying both the branch and the date/time. The branch specification must end in “`*`”, since it makes no sense to specify both a date/time and an exact version number. For clarity, you should separate the date/time from the version number with a semicolon.

Here are some examples of legal version specifications for the EBS and ES commands:

`23` selects version 23.

`25-Dec-92,13:00:00` selects the latest trunk version as of 1pm, Christmas day, 1992.

`@snapname.ext` selects the version in the snapshot.

`3-Feb-88` selects the latest trunk version as of midnight, February 3rd, 1988.

`*;3-Feb-88` equivalent to the previous example

`6.*;3-Jun-91` selects the latest branch version 6.x as of midnight, June 3, 1991.

`@filelist;12:00:00` selects the latest version as of noon today in the branch specified in the named file list.

You should be aware, when using retrieval by date/time, that the date/time which TLIB uses is the date/time-stamp that the source file had when it was saved in the library, not the time at which the library was updated (but see the `TOUCHSOUR` parameter, p. 288).

Comment Files

TLIB can read comments from a file when updating a library with a new version. This is useful, for example, if you wish to update many TLIB library files with new versions, and you want to enter the same comment for each file, but the comment will not fit on a single line (on the DOS command line). Suppose, for example, that `COMMENTS.TXT` contains the comments. You could make TLIB update the libraries and read the comments from `COMMENTS.TXT` like this:

```
  tlib uf *.c @comments.txt
```

Note that TLIB automatically stores the file name, date, etc. on the first comment line, which uses up quite a bit of the available room on the line. If the first line in `COMMENTS.TXT` will not fit in the space that remains, TLIB will automatically move all the comments down one line (as if you'd left a blank line at the top of `COMMENTS.TXT`).

You can use wildcards in the name of the comment file; TLIB will substitute for the wildcards from the name of the source file. Thus, a set of modified source files and associated comments can be prepared in advance, then the update operations can be done all at once. For example:

```
  tlib uf *.c @*.cmt
```

...which would read comments from `MYFILE.CMT` for source file `MYFILE.C`, and from `YOURFILE.CMT` for `YOURFILE.C`.

This is useful in highly “controlled” environments, where programmers can extract and check-out modules, but are not allowed to do “update” operations. Instead, the programmers must send the files to a “system librarian” to be checked-in. In a networked environment, the system librarian is the only person who requires read/write access to the library files; programmers do not need write access to the libraries if they have read/write access to the lock files (hint: for Novell and most other LANs, this is easily accomplished by keeping the library and lock files in separate directories, as shown in the example on page 260).

This type of control might be required by the configuration management plan for some U.S. government contracts.

As a shortcut which is equivalent to the "`@*.ext`" format, you can specify just "`@`" for the comment file, and TLIB will derive the comment file name from the source file name. By default, the name of the comment file is simply the name of the source file with the extension changed to `.CMT` (you can change this convention with the CmtExt configuration parameter, as described on p. 263). For example, if you give the command:

```
  tlib uf *.? @
```

...then the comments for `MYFILE.C` are expected to be in `MYFILE.CMT`, and the comments for `YOURFILE.H` should be in `YOURFILE.CMT`.

See also p. 105.

Configuration

Alphabetical listing of parameters

T - means TLIB uses this parameter
C - means CMPR uses it
D - means TLMERGE & DIFF3 use it

	parameter	default	page
T	AATTR *<set/preserve/reset>*	Set	287
T	ABORT *message*		309
T C D	ADDCTRLZ *<Y/N>*	N	276
T	ARCCMD *<path-of-pkpak.exe>*		328
T	ARCEXT *extension*	ARC	328
T	ARCTEMP *<temporary-directory>*		328
T	AUTOBRNCH *<Y/N/Q>*	Query	
T	AUTOSET *filename*	AUTOSET.BAT	292
T	BANNER *<1-42>*,"*string*"		333
C D	CMPRDETAB *<Y/N>*	N	339
C D	CMPRENTAB *<Y/N>*	N	339
T	CMTEXT *extension*	CMT	263
T	CMTFLAG *<1-80>,<quoted-string>*		289
T	CMTSUFFIX *<1-253>,<quoted-string>*		289
T	COLORIZE *<Y/N>*	N	313
T	COLOROFF *<string>*		315
T	COMMANDS *<comma-delimited-list>*		330
T	CREATETF *<Y/N>*	N	298
T	CZTRUNC *<Y/N>*	N	291
D	D3COLLIDE *<line-to-insert>*	/* ###...	340
D	D3FLAG2 *<0-253>*,"*string*"		340
D	D3FLAG3 *<0-253>*,"*string*"		340
T	DATAPATH *<Y/N>*	N	278
T	DEFEXT *extension*		263
T	DELETESRC *<Y/N>*	N	278
T	DETABE *<Y/N/Maybe>*	Maybe	255
C D	DIFFLINES *<100-16380>*	3000	339
T	DOTDOTOK *<Y/N>*	N	303

	parameter	*default*	*page*
T	`ELSE`		323
T	`ELSEWHERE` *<Y/N>*	`N`	308
T	`ENDIF`		321
T	`ENTABU` *<Y/N>*	`N`	256
T	`EQUALDATE` *<Y/N>*	`N`	275
T	`ERRORPAUS` *<0-3>*	`2`	312
T	`EXITPAUSE` *<Y/N>*	`N`	313
T	`EXTENSION` *ext1*`,`*ext2*`,`*...*		254
T	`FILETYPE` *<Auto/Binary/Text/Runlen/EOFtol>*	`AUTO`	294
T	`FIND1FILE` *<Y/N>*	`N`	
T	`FIXKEYWD` *<Y/N>*	`Y`	277
T	`FNAMECASE` *<U/L/A>*	`A`	309
T	`FORCEREFR` *<Y/N>*	`N`	306
T	`FORCEU` *<Y/N>*	`N`	275
T	`HELP` *<1-49>*`,"`*string*`"`		329
T	`ID` *name*		266
T	`IF` *<list-of-wildcard-specs>*		321
T	`IFF` *expression*		323
T C D	`INCLUDE` *path*		264
T	`JFILE` *<name-of-journal-file>*		190
T	`JOPTIONS` *<journal-options>*	`UOCAP` *or* `UIOAP`	190
T	`KEYFLAG` *<1-254>*`,"`*string*`"`		213
T	`LET` *name=expression*		270
T	`LEVEL n=`*name* `d=`*path* `p=`*name etc.*		300
T	`LIBDIRQ` *<1-54>*`,"`*string*`"`		
T	`LIBEXT` *extension*		262
T	`LOCKING` *<Y/N/B/W>*	`N`	265
T	`LOGFLAG` *<1-240>*`,"`*string*`"`		222
T	`LOGPREFIX` *<1-80>*`,"`*string*`"`		222
T	`LOGSUFFIX` *<20-253>*`,"`*string*`"`		222
T	`LOGTIME` *<Y/N>*	`Y`	266
T	`LOGUSER` *<Y/N>*	`N`	266
T	`LOGWIDTH` *<20-254>*	`79`	267
T	`LOKEXT` *extension*		262
T	`LONGNAMES` *<Y/N/M>*	`M`	77
T	`MAKEDIRS` *<Y/N>*	`N`	315
T	`MAXLINES` *<100-16380>*	`4000` or `16000`	
T	`MULTIPASS` *<Y/N>*	`Y`	335
T	`NEWLINE` *<CRLF/LF/CR>*	`CRLF`	311
T	`NT351BUG` *<Y/N>*	`N`	316
T	`NTFS35BUG` *<Y/N>*	`N`	317
T	`NUMBANNER` *<1-42>*	`0`	329
T	`NUMLIBDIR` *<1-5>*	`1`	329

	parameter	default	page
T	NUMHELP *<1-49>*	0	329
T	NUMPROMPT *<1-42>*	1	329
T	OLDDATE *<Y/N>*	Y	267
T	ONETHREAD *<Y/N>*	N	292
T	PASSSIZE *<100-16380>* *(this is a synonym of* MAXLINES*)*	4000 or 16000	
T	PATH *<path-of-libraries>*	=	257
T	PROJLEV *name*		299
T	PROMPT *<1-42>*,"*<string>*"		329
T	QUERIES *<Y/N>*	Y	269
T	QUIET *<Y/N>*	N	279
T	READONLY *<Y/N>*	N	279
T	READONLYB *<Y/N/W>*	N	280
T	READONLYT *<Y/N/W>*	Y	320
T	REFNEWLN *<CRLF/LF/CR>*	NEWLINE	
T	REFSUBDIR *<directory-name>*		305
T	RELAXVERS *<Y/N>*	N	307
	REM *or* ! *Anything*		252
T	REPLACE *<Y/N/Q/A>*	Q	268
T	REPLROBR *<Y/N/Q/W>*	N	282
T	ROLOCKS *<Y/N>*	N	280
T	SAY *message*		309
T	SERIALNO *vvv-sssss-nn-cccccccccccc*		
T	SET *name=<unquoted-string>*		270
T	SETFTIMEW *<Y/N>*	N	274
T	SHEIGHT *<0 or 8-70>*	25	284
T	SHOWLNAME *<Y/N>*	Y	
T	SLASHCONT *<Y/N/M>*	Y	285
T	SLICKEPSI *<Y/N/Maybe>*	Maybe	292
T	SWIDTH *<0 or 40-32765>*	80	284
T	TOPRELATI *<Y/N/Maybe>*	Maybe	301
T	TOUCHSOUR *<Y/N/Modified/Revhist>*	N	
T	TOUCHU *<Y/N>*	Y	275
T	TRACK *<Y/N/Maybe>*	N	297
T	TRACKEXT *extension*	TRK	308
T	TREEDIRS *<Y/N>*	N	300
T	UNARCCMD *<path-of-pkunzip.exe>*		326
T	UPDATENEW *<Y/N>*	N	276
T	USEDUPHAN *<Y/N/Maybe>*	Maybe	285
T	USEUMBS *<Y/N>*	Y	319
T	VALIDATE *<Y/N>*	Y	285
T	WARN *message*		309
T	WORKDEPTH *nn*	0	310

	parameter	*default*	*page*
T	`WORKDIR` *<path>*	`.\` *(usually)*	304

Configuration overview

TLIB supports a large number of configuration parameters to allow you to tailor its behavior to your needs. However, most of the configuration parameters have reasonable default values so that you do not need to worry about the features which you do not require.

To avoid making users contend with commands and features which they may not need, we have even made the TLIB user interface configurable in command-line versions of TLIB. That way, unneeded commands do not even appear on the menu.

TLIB Combo Edition comes with a Windows-based Configuration Wizard, and TLIB for DOS comes with a program called TLIBCONF, to help you set up your initial TLIB configuration file. *You should run the TLIB Configuration Wizard or TLIBCONF even if you upgraded to TLIB 5.50 from an earlier version of TLIB.* See p. 15.

Where is the Configuration File?

TLIB first looks for a file named "`TLIB.CFG`" in the current directory. If it is not found, TLIB checks the DOS environment area for a "`set tlibcfg=path\file.ext`" command. If there was one, then TLIB will try to open `path\file.ext` as the configuration file. If that fails, then TLIB will look in the directory from which it was run for a file named "`TLIB.CFG`".

If you want to prevent TLIB from looking for `TLIB.CFG` in either the current directory or the directory containing `TLIB.EXE`, then use "`set tlibcfg=!path\file.ext`" (the "`!`" means "force"). However, if you do this then TLIB will not run at all unless it finds the specified configuration file.

There is also a semi-secret "patch point" which can be used to force TLIB to look in one and only one place for its configuration file; see p. 105.

In addition, before reading the regular `TLIB.CFG` file, TLIB looks for and (if it exits) reads a supplemental configuration file called `TLIB.SER`, in the same direcectory that the TLIB program resides in. Usually, `TLIB.SER` contains only `serialno` parameters (but you can put other configuration parameters into it, too).

TLIB for Windows also uses a `TLIB.INI` file, to store user interface parameters (command-line TLIBs may read it as well).

By default, `TLIB.INI` resides in the same directory as the TLIB executables. However, if you have a multi-user TLIB license and installed TLIB into a shared network directory, then this is inappropriate. There should be a different `TLIB.INI` for each user. So, if TLIB is installed into a shared network directory, you should use the `TLIBINI` environment variable to tell TLIB where to find `TLIB.INI`. The easiest way to do this is to add a `SET` statement to your `autoexec.bat` (or OS/2 `config.sys`) file, e.g.:

```
SET TLIBINI=C:\
```

DOS Environment Space

If you get an "Out of environment space" DOS error in response to your SET command, then under DOS 3.2 (and later) you may wish to increase your environment space using the `/E` parameter of the SHELL command in your `CONFIG.SYS` file (your DOS manual has details).

To approximately triple your environment space under DOS 3.2 and later:

```
shell=c:\command.com c:\ /p /e:512
```

Checking the configuration: -c command-line option

TLIB has a facility to "dump" most of TLIB's configuration data into a file, so that you can see the result of the environment variable substitutions, include directives, etc..

The "`-cfilename`" option writes the configuration information to filename.

Example:

```
tlib -cmyfile.txt q
```

That will run TLIB, write the configuration data to `myfile.txt`, and then exit to the operating system. ("`Q`" is "quit").

Note: The configuration parameters that TLIB dumps are the ones that you can reference as pseudo-environment variables in the configuration file via the `%tlibcfg:`*parametername*`%` syntax.

Note that empty/do-nothing `if/endif` blocks in your configuration file will not appear in the dumped configuration data. For example, if you configured:

```
if *.c,*.h
   track y
endif
track n
```

...then the `if/endif` block does nothing, so TLIB ignores it.

What does the configuration file contain?

The TLIB configuration file is a normal DOS text file, which you can edit with any text editor. The initial configuration file should be created by running TLIBCONF and answering the questions which it asks you. However, TLIBCONF only configures the most common, basic configuration parameters, so you may need to make additions or changes by editing the configuration file manually.

Configuration files can contain comment lines, which must begin with "`!`" or "`REM `". The other thing they contain is configuration parameters, one per line, in the formats described below.

C command: Overriding Configuration Parameters

With the C command, you can specify or override any configuration parameter, either on the DOS command line or interactively. For example:

```
TLIB c "readonlyb N" eb myfile.c
```
(means disable read-only-browse mode, then do browse-mode extract of `myfile.c`)

```
TLIB c readonlyb_n eb myfile.c
```
(another syntax to do the same thing)

```
TLIB c readonlyb=n eb myfile.c
```
(yet another syntax to do the same thing)

Using an underscore (_) instead of a blank after the configuration parameter name is more convenient, since it avoids the necessity of putting simple configuration parameters in quotes. It only saves a couple of keystrokes when typing. However, it avoids problems when passing TLIB configuration parameters to `.bat` files, since DOS's command processor strips off quote marks in `.bat` file parameters.

The “=” (as a third alternative to a blank or “_”) is allowed because we've observed a tendency for users to get confused and use “=” instead of a blank. However, it has the same problem that quote marks have: when passed in a `.bat` file parameter, “=” gets stripped off by the DOS command processor.

The C command can be specified several times, if you wish, to change several configuration parameters:

```
TLIB c keyflag_0 c logflag_0 e myfile.c
```
(means disable keyword and revision history log insertion, then extract myfile.c)

Even the “`INCLUDE`” directive is allowed, so you can use it to specify a large number of configuration parameters, like this:

```
TLIB c include_special.cfg e myfile.c
```

The only things allowed in the configuration file which you cannot specify with the C command are the “`if`” and “`endif`” directives (this restriction can be circumvented by putting the `if` / `endif` block in a file and loading it with the C command and the “`include`” directive).

`EXTENSION` *ext1*`,`*ext2*`,`*...*

The **Extension** configuration parameter is used to tell TLIB what file extensions you will be using. If you configure TLIB with the TLIB Configuration Wizard, and tell it the source file types that you will be using, it will correctly configure the Extension parameter for you.

Example:

```
 EXTENSION c,h,asm,pas,,bat,cmd
```

Note #1: there must be no dots, spaces or wild-cards in the list.

Note #2: the double comma after "`pas`" indicates that some of your files may have no extension at all (e.g., "`makefile`").

If you do not configure `EXTENSION` (or if you configure it incorrectly), then TLIB's L, C and O wild-card search modes will not work under some circumstances; see note #5, below, for details (if you care).

Note #3: if you configure `DEFEXT` ("default extension"), then that default extension will also serve as your first `EXTENSION` setting.

Note #4: The next 5 paragraphs are a description of why TLIB needs the `EXTENSION` configuration parameter (you can skip this if you don't care):

TLIB needs the `EXTENSION` configuration parameter when performing wild-card searches of library or lock files (with the L, C and O search modes), so that it can deduce the names of the source files from the names of the library or lock files when you give a wild-card specification which does not already give the extension.

For example, suppose that "`LIBEXT ?$?????`" is configured (a common setting). Then if you store a source file called `MYFILE.PAS` into a TLIB library, the library will be named `MYFILE.P$S`.

Now, if you do the command, "`TLIB E *.PAS`", TLIB actually searches for library files named `*.P$S` (because the default search mode for the E command is "L"). However, TLIB can tell from the wild-card specification that you entered that the extracted files should have the "`.PAS`" extension, and it does not need the EXTENSION configuration parameter to figure out how to name the extracted files. When it finds the library file "MY-FILE.P$S", TLIB will deduce the first part of the source file name from the name of the library file, and will deduce the ".PAS" extension from your original wild-card specification.

However, if you give the command "TLIB E MYFILE.*", then when TLIB finds the TLIB library named MYFILE.P$S, it will not know how to name the extracted file. More precisely, the second character of the file extension cannot be deduced from either the name of the library file or the original wild-card specification. The extension could be .PAS, .PBS, .PCS, or whatever.

So, to resolve this ambiguity, TLIB consults the DEFEXT and EXTENSION configuration parameters: the first extension found which matches will be used. In this case, it would be the first extension found in which the 1st character is "P" and the third character is "S".

DETABE *<Y/N/Maybe>*

The **detabE** (DE-TAB on Extract) parameter determines whether tabs in the library file will be expanded to blanks in the source file during an E (extract) command. Setting this to Y will slow down these commands slightly (about 8%).

The default is DetabE M ("maybe," which means do not expand tabs unless EntabU Y was configured when the TLIB library file was created). Example:

```
 detabE N
```

By the way, TLIB is "case-insensitive," which means that upper-case and lower-case are interchangeable in most contexts. So you can specify "detabe n" or even "dEtAbE N" with the same effect.

Also, TLIB allows you to substitute an underscore, equal sign or tab character for the blank which normally follows the name of the configuration parameter. So you can specify "detabE_N" or "detabE=N" with the same effect. Using an underscore instead of a blank may be preferable when passing parameters to batch files, since "_" will be passed unmolested, but quotes and equal signs are removed from batch file parameters by the command processor.

Note that only 8-space tabs are supported by TLIB. If you use other tab stops, you should *not* set `DetabE`, `EntabU`, `CmprEntab` or `CmprDetab` to “`Y`” in your configuration file.

Also, note that tab conversions are not supported for files which contain lines that are longer than 254 characters, nor for `FILETYPE Binary` libraries.

The usual setting is the default, `DETABE MAYBE` (or just “`DETABE M`”), which means that tabs will be expanded when the source file is extracted if and only if blanks were converted to tabs when the file was stored (which is determined by how the `ENTABU` configuration parameter was set at the time the library file was created). (This avoids an anomalous situation in which changing the `ENTABU` and `DETABE` configuration settings could cause TLIB to find “changes” even in a freshly-extracted file.)

`ENTABU` *<Y/N>*

The **entabU** parameter is somewhat analogous to the detabE parameter, except that it determines whether blanks will be compressed to tabs when adding versions to the library file (with U and N commands). However, there is a subtle difference between commands which create a library file (e.g., N) and those which merely add another version to it. For those commands which add versions to an existing library file, the value of entabU used is the one that was set when the library file was created, rather than the current value. (The third byte of the library file will be `` `t' `` if entabU was N (false) when the library file was created.)

Note that tab conversions are not supported for files which contain lines that are longer than 254 characters, nor for `FILETYPE Binary` libraries.

Default is N (no, do not compress blanks to tabs in the library file). Setting this to Y will reduce the memory requirements of TLIB, and make some library files significantly smaller. Here we leave it at the default value:

```
 entabU N
```

`PATH` *<directory/folder path>*

The **path** parameter specifies the default path for libraries. It can be overridden by TLIB's `CP` (path) command. Default is “=”, which means the same directory as the source file. If you want library files to always be in the current directory (even when the source file is specified with a different subdirectory or drive), you can specify “`.\`” for the path.

You can, if you wish, keep your library files in the same disk directory as your source files. More often, however, library files are stored in a different hard disk directory, or perhaps on a network file server (or even a diskette).

TLIB provides the `PATH` parameter for this situation: it allows you to specify the path (drive and/or subdirectory) in which your library files reside. The most common form specifies the directory in which TLIB keeps the TLIB library files:

```
 PATH d:\path\
```

The CP command can be used to override the `PATH` configuration parameter, to specify a different subdirectory, using the usual DOS path specifications (see also p. 42). For example:

```
 TLIB CP B:\LIBRARY\ U D:MYFILE.TXT
```

TLIB will update the appropriate library file in directory `B:\LIBRARY\` with the latest version of `D:MYFILE.TXT`.

Note that if you neither configure the `PATH` parameter nor use the CP command to specify a library file path, then TLIB will expect the library file to be in the same directory as the source file. This is equivalent to configuring:

```
 PATH =
```

The CP command can also be used to specify the complete library file name, rather than allowing the library file name to be determined by the name of the source file. The file name must contain (or be followed by) a “.” (period), since this is how TLIB distinguishes the file name from the name of a directory.

This form of the CP command is seldom used, since it requires that you use a separate CP command for each library file. You will generally find it easier to let TLIB determine the library file name from the source file name, instead of using this form of the CP command. Nevertheless, here are a couple of examples:

```
TLIB CP MYLIB.XYZ
```

The library file is `MYLIB.XYZ`, in the same directory as the source file.

```
TLIB CP C:\LIBS\MYLIB.
```

The library file is `MYLIB` (with no extension), in directory `C:\LIBS`.

The `PATH` parameter and CP command have a third form, to handle a particular problem faced by some users. The problem is this: what do you do if you have two different source files with the same name but different extensions, and you want to have a library file for each of them? TLIB would normally try to use the same library file for both of them. C programmers sometimes face this: for example, they may need to have separate library files for `file.c` and `file.h`.

The simplest solution is to use the `LibExt` (and `LokExt`) configuration parameters (see p. 262).

An equivalent solution is to use the "`*.`*ext*" form of the `PATH` parameter or CP command to change the library file naming convention. Question marks ("`?`") can be used as wild cards in the new library file extension, to represent characters from the source file's extension.

Examples:

```
TLIB CP *.LIB
```

changes the library file extension to `.LIB`.

```
TLIB CP C:\LIBS\*.LIB
```

combines forms 1 and 3 of the CP command, to specify both a directory for library files (`C:\LIBS`), and a new library file extension.

```
TLIB CP *.?$? E FILE.C E FILE.H U FILE.BAT
```

extracts the latest version of `FILE.C` from a library file named `FILE.C$`, extracts the latest version of `FILE.H` from a library file named `FILE.H$`, and updates a library file named `FILE.B$T` with the latest version of `FILE.BAT`.

```
 TLIB CP *.?$ N FILE.C
```

creates a library file called `FILE.C$` for the source file `FILE.C`.

Note: the `LIBEXT` and `LOKEXT` configuration parameters provide a simpler mechanism for selecting the naming conventions for library and lock files. See p. 262.

If you are using check-in/out library locking, then there is a fourth variation of the `PATH` parameter and CP command available. (check-in/out locking is used for multiple-programmer projects; it is explained starting on p. 97.)

Along with both the second and third forms of the CP command, you can specify an alternate extension or path for the lock file. This is only meaningful if you have check-in/out locking enabled. The lock file extension and/or path is separated from that of the library file with a slash character (`/`).

There are several circumstances under which you would need to specify the extension or path of the lock file:

1) If you have different files with the same names but different extensions (e.g., `file.c` and `file.h`), then you would need to change *both* the library file extension and the lock file extension to depend upon the source file extension.

2) If you keep your TLIB library files on an WORM (write-once optical) disk drive, such as the IBM 3363, then keeping lock files on a conventional (magnetic) disk drive would reduce the consumption of WORM storage. TLIB is ideally suited for use with WORM drives, since library files are only appended, not replaced, when a new version is added. With other version control systems, the entire library file is replaced, wasting an extravagant amount of WORM disk storage.

3) If you keep your TLIB library files in a network file server directory for which most users are not allowed delete access, then it is advantageous to

put the lock files in a different directory so that TLIB can delete useless lock files when a version is checked-in. (TLIB's check-in/out locking mechanism will still function correctly even if it cannot delete obsolete lock files, but the obsolete lock files will needlessly clutter your disk.)

4) In highly “controlled” environments, programmers can extract and check-out modules, but might not be allowed to do “update” operations. Instead, the programmers must send the files to a “system librarian” to be examined and checked-in. To support this environment, the library files can be kept in a network file server directory to which only the system librarian has write access, but the lock files would be kept in a directory to which the programmers have both read and write access. This type of control might be required by the configuration management plan for some U.S. government contracts (see also pp. 244 and 105).

Configuration file example:

```
 PATH D:\LIBS\/D:\LOCKS\
```

configures TLIB to put its library files in `D:\LIBS\`, and its lock files in `D:\LOCKS\`.

“CP” command examples:

```
 TLIB CP C:\LIBS\*.?$?/?^? U A:FILE.PAS
```

checks in and updates the source file `A:FILE.PAS` to library file `C:\LIBS\FILE.P$S`, with lock file `C:\LIBS\FILE.P^S`. The `PATH`, `LIBEXT` and `LOKEXT` paramters are overridden.

```
 TLIB CP D:\LIBS\*.?$?/C:\LOCKS\*.?^? U A:FILE.C
```

checks in & updates source file `A:FILE.C` to library file `D:\LIBS\FILE.C$`, with lock file `C:\LOCKS\FILE.P^S`.

A path specified as “=” is special. It indicates that the library (and/or lock) files are to be found in the same directory as the source files, even if you have specified a source file which is not in the current directory. Note that this differs subtly from the default, which is "`.\`", which means the current directory; the two library path specifications indicate different directories when the user specifies a source file with an explicit path (that is, a file not in the current directory).

Using the `PATH` configuration parameter or CP command, you can specify several library directories separated by semicolons, and TLIB will check each directory in turn for your TLIB library file(s).

This is useful when you use more than one directory to store your TLIB libraries. One common use is to let you keep a different directory of TLIB libraries for each program or project, plus one or more "common" directories of shared modules. For example:

```
PATH f:\deptlibs\;f:\corplibs\
```

TLIB will first look for a library file in directory `f:\deptlibs`, and then (if it wasn't found), in `f:\corplibs\`.

Another example:

```
PATH f:\deptlibs\;=;.\;..\
```

TLIB will first look in the `f:\deptlibs\` directory. If the library file is not found there, TLIB will check the directory containing the source file (=). If the library is not there, either, TLIB will try the current directory (`.\`), which may or may not be different from the directory containing the source file. Finally, if a library file still has not been found, TLIB will check the "parent" directory (`..\`).

If you are using check-in/out locking (p. 265), and if you keep lock files in different directories from the library files, you must specify the lock file directories, too. The lock directories are in 1:1 correspondence with the library directories; if one or more of the lock directories is omitted, then the corresponding library directory is used, instead. For example:

```
PATH f:\deptlibs\;f:\corp\/f:\deptloks\
```

is equivalent to:

```
PATH f:\deptlibs\;f:\corp\/f:\deptloks\;f:corp\
```

With either of these `PATH`s, if the library was found in `f:\deptlibs`, then the lock file will be placed in `f:\deptloks`, but if the library was found in `f:\corp`, then the lock file will also be in `f:\corp`.

`LIBEXT` *<extension>*

The **libext** parameter is used to specify a naming convention for TLIB library files. It can be overridden by the "`*.ext`" form of TLIB's CP (path) command. The default is "`TLB`", which means that library files will have the same name as the corresponding source files, except that the extension will be `.TLB`. If you will need to maintain TLIB libraries for two source files with the same name but different extensions (e.g., `file.c` and `file.h`), then the default naming convention will not work, since both source file names would "map to" the same library file name. You should use the `LibExt` parameter to make the library file's extension depend upon the source file's extension: if you specify a "`?`" in the *<extension>*, then the corresponding letter from the source file extension will be used in the library file extension. For example, if you configure

```
 LibExt ?$?????
```

then for source file `quarkle.c`, the library file would be named `quarkle.c$`; for source file `gronk.pas`, the library file would be named `gronk.p$s`, etc.. This is the most common setting for `LIBEXT`.

Note: No *not* specify `LIBEXT` or `LOKEXT` within an `IF`/`ENDIF` block in your configuration file.

Note to users of Opus Make: For some additional advice on using TLIB's `LibExt` configuration parameter, please refer to the description of the `TLIBSUFFIX` macro in your Opus Make manual (look for "tlibsuffix" in the index).

`LOKEXT` *<extension>*

The **lokext** parameter is used to specify a naming convention for TLIB lock files. It is exactly the same thing as LibExt, except that it affects the naming of lock files instead of library files. The default lock file extension is "`LOK`". If you configure

```
LokExt ?^?????
```

then for source file `quarkle.c`, the lock file would be named `quarkle.c^`; for source file `gronk.pas`, the lock file would be named `gronk.p^s`, etc..

`CMTEXT` *<extension>*

The **cmtext** parameter is used to specify a naming convention for comment files. It allows the use of a shortcut form when telling TLIB to read version comments from a text file. The format of the CmtExt parameter is the same thing as for LibExt and LokExt, but it affects the naming of comment files instead of library files and lock files. The default is

```
cmtext cmt
```

Example: suppose you configure

```
CmtExt ?CM
```

...and issue the command

```
tlib u gronk.pas @
```

then TLIB would expect the comments to be in a file named `gronk.pcm`.

`DEFEXT` *<extension>*

The **defext** parameter is used to specify default file name extension for your source files. For instance, a COBOL programmer might use source files which all end in the extension "`.COB`". He could avoid having to always type "`.COB`" when telling TLIB the name of a text file by adding the following line to his TLIB configuration file:

```
DefExt COB
```

Similarly, a Pascal programmer could configure:

```
 DefExt PAS
```

You can still specify an explicit file name extension whenever you wish, and if you need to specify a text file which has no extension at all, just end the name with a period (“.”).

For example, if you configure the default extension as “`DefExt PAS`”:

MYFILE.C means MYFILE.C
MYFILE means MYFILE.PAS
MYFILE.PAS means MYFILE.PAS
MYFILE. means MYFILE

Our thanks to Mr. Phil Gerber for suggesting this feature.

`INCLUDE` *<filename>*

The **include** parameter allows configuration files to “call” one another. This is handy if you need to have several different configuration files but don't want to duplicate most of the configuration file contents. For instance, you might want to change just the default library path, like this:

```
 REM - just like “regular” TLIB configuration except
 REM - use Tom's private directory for TLIB libraries.
 INCLUDE G:\REGULAR.CFG
 PATH c:\tlib\
```

One common use for the include parameter is in a networked environment, where each workstation (or each “project” directory on a particular workstation) might need *almost* the same configuration parameters as every other. You could set the `tlibcfg` environment variable to point to a “standard” configuration file, “`set tlibcfg=f:\standard.cfg`”. Then each workstation (or project directory) for which configuration changes are required could contain a short `tlib.cfg` file which begins, “`include f:\standard.cfg`”, and then changes whatever configuration parameters need to be altered.

You can nest `INCLUDE`d configuration files to a depth of 3 (or 4, if you count the main configuration file).

`LOCKING` <*Y/N/B/W*>

The **locking** parameter changes the behavior of the E (extract) and U (update) commands. When locking is enabled, the E command causes the library to be "checked-out" to the current user; if the library is already checked-out to someone else, the E command will fail. When locking is enabled, the U command will cause the library to be "checked back in" after it is updated with the new version; the library must have been already checked-out to the current user, otherwise the U command will fail.

Note: even if the update aborted because there were no changes to the source file, the library will still be checked back in. See pp. 97 -104.

If you enable locking, you'll also want to use the `PROMPT`, `HELP` and `COMMANDS` parameters to customize the user interface (p. 329).

Default for the locking parameter is N, check-in/out locking disabled. You can enable it like this:

```
 locking Y
```

TLIB 5.0 added support for two new locking modes: "`Weak`" and "`Branch/level`". They are for use when `LOCKING Y` (full locking) is too constraining because you need to be able to have multiple programmers working on a single module at the same time. See "*Weak Locking and Branch/Level Locking*" (p. 98) for details.

Warning: "`LOCKING B`" (branch/level locking) should only be used with project levels for which `s=N` (or `p=`*something*) has been configured in the `LEVEL` configuration parameter, since only the current project level will be locked.

`ID` <*name*>

The **ID** parameter specifies the user name, which is needed to implement check-in/out locking. The configured ID overrides the DOS "`set tlibid=`*name*" command, and can be overridden by TLIB's CW (who) command. Note that there must *not* be spaces surrounding the "=" in "`set tlibid=`*name*". See pp. 89 & 100.

Example:

```
 id Dave
```

`LOGUSER` *<Y/N>*

The **loguser** parameter specifies whether or not the user id should be included in the version definition comment lines. The default is now `LOGUSER Y`, which is changed from TLIB 4.12.

There's probably no good reason to change it.

```
 loguser Y
```

`LOGTIME` *<Y/N>*

The **logtime** parameter can be set to N if you want only the date of the source file to be included in each version definition comment line. Normally, both date and time are included. Set logtime to N if you do not want the time included. The default is Y.

Normally, you should not configure `LogTime N` unless you also configure `OldDate N`, lest TLIB be unable to correctly set the timestamp of extracted source files. If both `OldDate Y` and `LogTime N` are configured, TLIB will create the files with the proper (old) date, but a time of 0:00.

Also, you should not configure `LogTime N` if you want to be able to retrieve old versions by date/time (p. 241).

```
 logtime Y
```

`OLDDATE` *<Y/N>*

When TLIB extracts a source file from its library file (using the E command), the source file is normally created with the original date and time that it had when it was added to the library.

If you prefer that TLIB re-create source files with the current date and time, use the **OldDate** parameter. If you configure `OldDate N`, then TLIB will create extracted source files with the date and time set to “now.” You may wish to do this, for instance, so that MAKE will correctly rebuild object files which depend upon the extracted source files.

The default is Y (yes), create extracted files with the original (old) date:

```
 olddate Y
```

See also pp. 276 , 358 and 366.

`LOGWIDTH` *<20-254>*

The **logwidth** parameter specifies the maximum length of a version comment line. Default is 79, for display on an 80-column monitor. You might want to use a smaller value to avoid truncation if you are using logflag to insert the version log into your source file, since the version comment lines will be truncated as needed to fit between the logprefix and logsuffix.

Note that the first comment line also contains the file name, date, time and/or user-ID, so there may not be much room left for comments on the first comment line if logwidth is small. The example logprefix and logsuffix leave columns 2 through 78 available for the version comment line, so we'll set logwidth to 77 (the example is on p. 224).

```
 logwidth 77
```

See also `SWIDTH`, p. 284.

`LOGFLAG` *<1-254>*`,"`*<string>*`"`
`LOGPREFIX` *<1-80>*`,"`*<string>*`"`
`LOGSUFFIX` *<20-254>*`,"`*<string>*`"`

The **logFlag**, **logPrefix** and **logSuffix** configuration parameters are described under “Revision History Logging,” p. 222.

`KEYFLAG` *<1-254>*`,"`*<string>*`"`

The **keyFlag** parameter is described under “Keywords,” p. 212.

`REPLACE` *<Y/N/Q/A>*

The **replace** parameter specifies whether or not an extract (E, EB, etc.) command will replace the source file if it already exists. Default is Q (query user before replacing). If you run TLIB from within DOS batch files and don't want TLIB asking questions at unpredictable times, you can specify N (no, abort the command if the source file already exists). Changing this parameter to Y is dangerous, since you could accidentally delete your latest source code file if you used the wrong TLIB command. We recommend that you leave this set to Q (the default), or set it to N (abort if file already exists), like this:

```
 replace N
```

TLIB 5.50 supports four different `REPLACE` modes when extracting files. The fourth, `REPLACE ABSOLUTELY` (or `REPLACE A`) was new to TLIB 5.0. `REPLACE A` is like `REPLACE Y` except that there is no warning when it is used while processing multiple commands using wild-cards. This is a *dangerous* configuration option; do not use it unless you really need to.

The `REPLACE Y` option is also dangerous, but if you extract multiple files using wild-cards or a file-list, it will give you one chance to bail out:

```
 Warning!  "REPLACE Y" configuration parameter active!  Continue?
```

`REPLACE A` suppresses even this warning.

The default is still `REPLACE Q`, which queries you before replacing files.

Note that the `REPLACE` configuration parameter interacts with two other configuration parameters: `REPLROBR` and `QUERIES`.

`QUERIES` *<Y/N>*

The **Queries** parameter lets you prevent TLIB from prompting you with Yes/No questions. Configuring `QUERIES N` prevents TLIB from pausing to ask such questions. More specifically, it effectively forces a "No" answer to all such questions.

This feature is intended for use in situations where it is not appropriate to ask the user a question (e.g., in some batch applications or when TLIB is being driven by another program).

Since TLIB's yes/no questions are generally used for "This might be dangerous, are you sure?" kinds of situations, forcing a "No" answer rather than a "Yes" answer) is the conservative choice.

When `QUERIES N` is configured, the yes/no questions will still be displayed, but the user will get no chance to answer them.

Note the relationship between `QUERIES N` and the `REPLACE` configuration parameter. If you configure `QUERIES N`, then `REPLACE Q` (the default) and `REPLACE N` are almost equivalent: the only difference is that with `REPLACE Q` configured, the question:

filename `already exists.  Replace it?`

...will still be displayed (though you won't get a chance to answer it), but with `REPLACE N` configured, the question will be suppressed altogether. (See the description of `REPLACE`, p. 268.)

`SET` *name=unquoted-string*

You can use `SET` configuration parameters to define pseudo-environment variables, which you can reference in the TLIB configuraton file via the `%`*name*`%` (or `%!`*name*`%` or `%!!`*name*`%`) syntax, just like real environment variables. The syntax is the same as the DOS and OS/2 "`set`" command:

```
SET name=string
```

The `SET` configuration parameter overrides any normal environment variable setting for the same name, and the name is case-insensitive.

There is no default for this configuration parameter.

See also "environment variables..." (p. 80), and the `LET` parameter (below).

`LET` *name*=<*expression*>

The `LET` configuration parameter is a variant of the `SET` parameter. The difference is that `LET` evaluates the right hand side of the assignment as an expression before assigning it. The syntax is:

```
LET name=<expression>
```

where <*expression*> is an expression consisting of literal integers, short literal strings, operators, and parentheses. Literal strings can be surrounded by either `'single'` (apostrophe) or `"double"` quote marks, and can be up to 80 characters long (including the quote marks).

The following unary operators are supported:

operator	result type	operation
`-`	integer	negate an integer
`NOT`	0 or 1	true if operand is zero
`LC`	string	convert a string to lower-case
`UC`	string	convert a string to upper-case
`UNQ`	string	"unquote" - remove the quotes from a string
`LEN`	integer	length of a string (including quotes, if any)
`SIZ`	integer	size of a file, or -1 if missing
`NAM`	string	"truename" - expand file name into full path

The following binary operators are supported:

operator	result type	operation
`+`	integer	addition of two integers
`-`	integer	subtraction
`*`	integer	multiplication
`/`	integer	division
`MOD`	integer	remainder
`AND`	0 or 1	“1” if both operands are non-zero
`OR`	0 or 1	“1” if either operand is non-zero
`<`	0 or 1	numeric "less than" test
`<=`	0 or 1	numeric "less than or equal" test
`>`	0 or 1	numeric "greater than" test
`>=`	0 or 1	numeric "greater than or equal" test
`==`	0 or 1	numeric equality test
`<>`	0 or 1	numeric inequality test
`EQI`	0 or 1	string equality test (case-insensitive)
`EQ`	0 or 1	string equality test (case-sensitive)
`NEI`	0 or 1	string inequality test (case-insensitive)
`NE`	0 or 1	string inequality test (case-sensitive)
`.`	string	string concatination
`SST`	string	substring
`SPL`	string	“split” - get specified part(s) of a path\name

Evaluation of binary operators is strictly left-to-right, and all binary operators have the same precedence. All unary operators have the same precedence, too, but unary operators have higher precedence than binary operators.

Note that multiplication and division do *not* have higher precedence than addition, subtraction, and the compare operators. Thus, for example:

Expression	Means	Evaluates to
`1+2*3`	`(1+2)*3`	`9 (`*not* `7)`
`1 < 2*3`	`(1<2)*3`	`3 (`*not* `1)`
`- 1 + 2`	`(-1)+2`	`1 (`*not* `-3)`

Strings and integers are generally interchangable. You can, for instance, concatinate an integer to a string, and if a string contains all numeric characters (and perhaps a leading minus sign) then you can use the string in arithmetic expressions as an integer. TLIB will convert automatically between strings and integers, as necessary.

To force conversion of a number to a string, concatinate the number to a 0-length string, like this:

```
let quoted120= ″ . (12*10)
```

The result of the concatination has the quote-type of the left-hand operand (apostrophes, in this example). To force conversion of a string to a number, add zero to it or multiply it by 1, as in this example:

```
let numeric120= 0 + ('12' . '0')
```

To include a quote mark in a string, quote it with the other kind of quote mark. For example, the following example prints the same line twice. Note that the expression assigned to `MOTTO` contains both an apostrophe and two double quote marks. (The `UNQ` operator and the `SAY` configuration parameter are explained below.)

```
let motto=UNQ ("Picard/Riker in '96." . ' "Make it so!"')
say %motto%
say Picard/Riker in '96. "Make it so!"
```

Most of the operators have obvious functions. However, a few of the operators need some explanation.

The `UNQ` operator "unquotes" a string. The result is still a string, but without the quote marks. Since every literal string includes the surrounding quote marks as part of the string, if you want a string without the quote marks then you must use the `UNQ` operator.

The `LEN` operator returns the length of a string, including the quote marks, if any. If you want the length without the quote marks, then use `LEN UNQ` (expression). For example:

```
let x=12345
let y=(len unq '%x%') - 1
say x=%x%, Log10(x)=%y%.something
```

The `SST` ("substring") operator returns a specified portion of a string.

The left operand of `SST` is the input string. The right operand is a string containing a pair of integers, separated by a colon, like "0:1", or else a single integer. The first (or only) number is the subscript of the first character to be returned; the characters are numbered from the left starting with 0, and also from the right starting with -1. The second number (if any) is the desired maximum length of the result.

Quote marks are not automatically included in the result. Beware of the fact that, using the `SST` operator, you can easily create strings that have a quote mark at only one end of the string.

Examples:

```
"abcdef" SST '1:2'                = ab
"abcdef" SST '-7:2'               = ab
"abcdef" SST '0:3'                = "ab
(UNQ "abcdef") SST '0:2'          = ab
″ . ("abcdef" SST '1:2')         = 'ab'
(UNQ "abcdef") SST '1:2'          = bc
(UNQ "abcdef") SST 1              = bcdef
(UNQ "abcdef") SST '2:2'          = cd
(UNQ "abcdef") SST 2              = cdef
(UNQ "abcdef") SST '2:999'        = cdef
(UNQ "abcdef") SST -2             = ef
(UNQ "abcdef") SST '-2:1'         = e
```

The `NAM` ("truename") operator expands a file name into fully-qualified path and name. For example, if your current directory is c:\work\hdr\ then:

```
NAM "myfile.x"                    = "c:\work\hdr\myfile.x"
nam "..\myfile.x"                 = "c:\work\myfile.x"
```

Beware that some network software has bugs that prevent the `NAM` operator from working correctly for files on network drives.

The `SPL` (split path\filename) operator splits a DOS or OS/2 directory path into parts, at the "\"s, and returns the specified parts. The drive letter or "\\server\volume\" part is part number 0. The rest of the parts are numbered two ways: with negative numbers, from right to left; and with positive numbers, from left to right. Leading and trailing slashes are removed, except on part 0. If a range of parts is specified, then the intervening slashes are included.

The left operand of `SPL` is a string containing the path to be split. The right operand is a string containing a pair of integers, separated by a colon, like "0:1". As a shorthand, if both numbers are the same (only one part of the path\filename is desired), you can just specify a single integer for the right operand. E.g., specifying '0' for the right operand is the same as specifying '0:0'.

Examples:

```
 "d:\aaa\bbb\ccc" SPL '0:0'                 = "d:\"
 "d:\aaa\bbb\ccc" SPL '0'                   = "d:\"
 "d:\aaa\bbb\ccc" SPL 0                     = "d:\"
 "d:\aaa\bbb\ccc" SPL "-1:-1"               = "ccc"
 "d:\aaa\bbb\ccc" SPL -1                    = "ccc"
 "d:\aaa\bbb\ccc" SPL 3                     = "ccc"
 'd:\aaa\bbb\ccc' SPL 3                     = 'ccc'
 UNQ ('d:\aaa\bbb\ccc' SPL 3)               = ccc
 (UNQ 'd:\aaa\bbb\ccc') SPL 3               = ccc
 (UC UNQ 'd:\aaa\bbb\ccc') SPL 3            = CCC
 "d:\aaa\bbb\ccc" SPL -3                    = "aaa"
 "\aaa\bbb\ccc" SPL "0"                     = "\"
 "aaa\bbb\ccc" SPL 0                        = ""
 "d:\aaa\bbb\ccc\" SPL 1                    = "aaa"
 "d:\aaa\bbb\ccc\" SPL -2                   = "bbb"
 "d:\aaa\bbb\ccc\" SPL 2                    = "bbb"
 "d:\aaa\bbb\ccc\" SPL '2:1'                = ""
 "d:\aaa\bbb\ccc\" SPL '1:2'                = "aaa\bbb"
 "d:\aaa\bbb\ccc\" SPL '-1:-2'              = ""
 "d:\aaa\bbb\ccc\" SPL '-2:-1'              = "bbb\ccc"
 "\\servr\sys\aaa\bbb\ccc\" SPL '-2:-1'     = "bbb\ccc"
 "d:\aaa\bbb\ccc\" SPL '0:-1'               = "d:\aaa\bbb\ccc"
 "d:\aaa\bbb\ccc\" SPL '0:-2'               = "d:\aaa\bbb"
 "d:\aaa\bbb\ccc\" SPL '0:1'                = "d:\aaa"
 "\\srvr\sys\aaa\bbb\ccc\" SPL 0            = "\\srvr\sys\"
 "\\srvr\sys\aaa\bbb\ccc\" SPL '0:-3'       = "\\srvr\sys\aaa"
 "\\srvr\sys\aaa\bbb\ccc\" SPL -3           = "aaa"
 "\\srvr\sys\aaa\bbb\ccc\" SPL -1           = "ccc"
 "d:\aaa\bbb\ccc\" SPL '2:-1'               = "bbb\ccc"
 "d:\aaa\bbb\ccc\ddd" SPL '2:-1'            = "bbb\ccc\ddd"
 "d:\aaa\bbb\" SPL '2:-1'                   = "bbb"
 "d:\aaa" SPL '2:-1'                        = ""
```

`SETFTIMEW` *<Y/N>*

The `SETFTIMEW` configuration parameter is mainly for use when running TLIB under MS-DOS in an OS/2 VDM. Configuring `SETFTIMEW Y` tells TLIB that a file must be open with write access if its date/time is to be changed.

```
 SETFTIMEW N   -  normal
 SETFTIMEW Y   -  write access is required to set a file's date/time
```

Note: TLIB will tell you if you should configure this parameter.

`FORCEU` *<Y/N>*

The **forceU** parameter specifies whether a U (update) command will be allowed to complete even if there have been no changes to the source file. Default is `N` (no, abort the command if there have been no changes).

```
 forceU N
```

`TOUCHU` *<Y/N>*

The **touchU** parameter specifies whether a U (update) command will cause the library file's date/time to be changed to the current date/time even when the operation aborts with the “No changes” message. If `TOUCHU` is `N`, then the date/time will not be changed unless the library file is actually extended with a new version. If `TOUCHU` is `Y`, then the date/time will be updated even if there were no changes in the current version (but the library file's archive attribute will be set only if a new version was actually added).

If you intend to use the F (fast Update) command, or if you use MAKE to update your libraries, you will want to leave `TOUCHU` set to `Y` (the default). Otherwise, you can set it to `N` (no, don't set date/time unless the library file was actually changed) without ill effects. Here we specify the default:

```
 touchU Y
```

`EQUALDATE` *<Y/N>*

The **EqualDate** parameter affects the date/time-stamp of a library file after has been updated. Normally, the date/time is set to “now” (the time when the library was updated). However, if you configure `EQUALDATE Y`, then the date of the library will be the latest of: (1) the old library file date, and (2) the source file date.

This normally causes the library file date to be exactly the same as the source file's date (since you don't usually update a library file with an old source file).

This configuration parameter is provided mainly for network users who use MAKE to extract the latest version of the source code whenever someone else has added a new version to the library, but who still wish to use the F (fast update) command to update their library files. By setting `EQUALDATE Y` and `OLDDATE Y`, you ensure that the source and library file dates are equal, so that both kinds of operations will work properly.

`UPDATENEW` *<Y/N>*

The **UpdateNew** parameter can be set if you would like to have one combined command do the functions of both U (update) and N (new). If you configure `UpdateNew Y`, then the U command will not fail due to a nonexistent library file; instead, it will create the library file, as if you had done the N command instead of U. Default is `N` (the U command reports an error if the library file doesn't exist).

```
 UpdateNew N
```

`ADDCTRLZ` *<Y/N>*

The **addctrlZ** parameter specifies whether or not TLIB will add a Ctrl-Z (end-of-file) character to the end of text files. This parameter affects TLIB, CMPR and DIFF3.

This parameter defaults to `N` (no, do not add the ctrl-Z), which is fine for most users. However, a few older programs may require the ctrl-Z; if you use such a program, configure:

```
 addctrlZ Y
```

`FIXKEYWD` *<Y/N>*

Whenever you extract a file containing keywords or a revision history, TLIB inserts the correct, up-to-date keyword or revision history log. However, when you store a new version with the U (update) or N (new library) command, it is optional whether TLIB will modify your source file to update the keyword or revision history information. The **FixKeyWd** parameter lets you tell TLIB whether you want this done.

By default, when you store a source file into a TLIB library with the U or N command, TLIB will also “fix up” any keyword information or revision history log in your source file, to make it reflect the new version number.

In TLIB 5, this now works even with multipass library files (which was not the case in TLIB 4.12). However, it is necessarily slower than with single-pass library files, since for single-pass libraries TLIB can write out the “fixed” source file from RAM, but for multipass libraries TLIB must do a full re-extract.

For files of modest size, this is fast and helpful, ensuring that your embedded keyword information is correct.

Having correct keywords is a real advantage, not just an aesthetic one, since TLIB 5's keyword-based version number checking means that incorrect `%v` keywords will cause warning messages to be displayed, and these warnings may be misleading if the keywords are incorrect. (This “keyword-based version number checking” is very useful: it helps you avoid lost changes even when locking is disabled or someone subverts the locking protections; see p. .)

Even if `FIXKEYWD Y` is configured, the fix-up is not done unless it is necessary. If there are no keywords and no revision history log in the source file, or if `KEYFLAG` and `LOGFLAG` are not configured, or if the source file will be deleted anyhow because you've configured `DELETESRC Y`, then the fix-up will not be done.

When the fix-up is done, however, it slows down the U or N command because TLIB must re-write your source file. The slowdown is slight for files of modest size, but it may be substantial for large files being stored in multipass TLIB libraries.

To save time by preventing TLIB from fixing your keywords, you can configure `FIXKEYWD N`. However, if you use the `%v` keyword, this is likely to result in bogus warning messages when you do another U command to store another version, due to the `%v` version number in your source file being incorrect.

The default is `FIXKEYWD Y` (note the change from TLIB 4.12).

See keywords (p. 212), revision history log (p. 222), `AATTR` (p. 287), and `TOUCHSOUR` (p. 288).

`DELETESRC` *<Y/N>*

The **DeleteSrc** parameter can be set to Y if you want the source file to be automatically erased after the U (update) command completes, unless an error occurs while updating the library file. Note that if the U command aborts because there were no changes, the source will still be deleted.

The `DeleteSrc` parameter affects only the behavior of the U (update) and N (new library) commands without the "B" (browse mode) or "K" (keep locked) options. The UK and UF commands are unaffected. The default for `DeleteSrc` is `N`, do not erase the source file.

This configuration parameter is unchanged from TLIB 4.12, except that the N (new-library) command now respects it, just like the U (update) command.

Here we leave it set to the default:

```
 DeleteSrc N
```

`DATAPATH` *<Y/N>*

The **DataPath** parameter should be set to Y if you use a "data path" resident extension to DOS, such as the one supplied with Novell's Netware or the DOS `APPEND` command. A data path extension lets you specify a list of default directories for data files, just as the regular DOS "path" command lets you specify a list of default directories for programs.

Many data path extension utilities work only on input data files. This can be a nuisance for TLIB, since TLIB will sometimes attempt to open a file first for input, then later for output or appending. If the file exists in another directory, but not in the current one, an open for input would succeed, but not an open for appending. Also, TLIB sometimes tries to open a file for input, in order to verify that the file does not already exist, before trying to create it; a data path extension can interfere with this test.

If you use a “data path” DOS resident extension, then you can configure TLIB to take this into account when opening files. Simply configure “`DATAPATH Y`”. This causes TLIB to always specify an explicit path when opening files (e.g., “`.\fname.ext`” instead of just “`fname.ext`”), which often has the salutary effect of preventing data path extensions from working their magic. Here, we leave it to the default.

```
 DataPath N
```

`QUIET` *<Y/N>*

Configuring “`QUIET Y`” will somewhat reduce TLIB's verbosity. It still won't satisfy the ex-Unix people (most of whom would prefer to see nothing but error messages), but it does cut the “noise level” somewhat.

Note that specifying “`-q`” as the first parameter to TLIB has the same effect, except that “`-q`” additionally suppresses the copyright banner. See also p. .

The default is “not quiet”:

```
 QUIET N
```

`READONLY` *<Y/N>*

The **ReadOnly** parameter instructs TLIB to keep the library files' “read-only” DOS file attribute set (except during an Update). This can help to prevent accidental erasure of the library files. The default is N, do not keep library files as read-only. Here we leave it set to the default.

```
 ReadOnly N
```

Note that the DOS `attrib` command can be used to manually set or reset the read-only file attribute. For example, if “`LIBEXT ?$?`” is configured, then the following command will make all the TLIB library files in the current directory read-only, by setting the read-only attribute “on”:

```
 attrib +r *.?$*
```

Similarly, this command will reset the read-only attributes to “off”:

```
 attrib -r *.?$*
```

`ROLOCKS` *<Y/N>*

As described above, if you want TLIB libraries to be stored as read-only files, you can use the ReadOnly option. However, TLIB's “lock” files (used for check-in/out locking) are not normally read-only, even if `READ-ONLY Y` is configured.

One of our customers needed to have the lock files read-only, too, so we added the **RoLocks** configuration parameter.

If you want lock files to be kept read-only (to avoid accidental deletion), configure:

```
 locking Y
 readonly Y
 rolocks Y
```

Note that `RoLocks` has no effect unless `LOCKING Y` and `READONLY Y` are also configured.

`READONLYB` *<Y/N/W>*

The **ReadOnlyB** parameter tells TLIB whether to set the “read-only” file attribute for “browse mode” source files. If you configure `READONLYB Y` then browse-mode files will be set to read-only after an EB command;

likewise, after you check-in/update a file with the U command, it will be changed to read-only, since you no longer have the file checked-out for modification. This is the most popular way to tell the difference between files which you have checked-out for modification and those which were extracted in browse mode.

This parameter is only useful if you are using check-in/check-out locking, primarily in networked environments (using the Network Version of TLIB).

A disadvantage of this approach is that you'll have to use the DOS “attrib -r” command to clear the read-only attribute before you can delete a browse-mode file.

In some cases, you might want to configure `READONLYB W` ("read-only browse mode files only in Work directory"), to make browse-mode files read-only in your work directories, but leave reference copies in project level reference directories writable. `READONLYB W` is for users who have a network that does not permit them to make files on the server read-only (or which does not allow other users to make them writable again). The most common example is a Unix-based file server, but one user reports having this problem with a Windows-NT 4.00 server used with NT 3.51 workstations.

See also: `READONLYT`, `REPLROBR`.

If you are using a local area network, and if your text editor warns you when it *loads* a read-only file (rather than when it *saves* the file), then you should use this configuration parameter. Otherwise, you may prefer using the `%l` keyword to warn you when you start to edit a file which is not checked-out (see p. 216).

Note that TLIB's T (test lock status) command also provides an easy way to determine which files you have checked-out for modification (see p. 102).

See also the **ReplRoBr** configuration parameter, below.

The default is N, do not make the browse-mode source files read-only. Here we leave it set to the default.

```
 ReadOnlyB N
```

Note: The `READONLYB` configuration parameter is unchanged from TLIB 4.12, except that `READONLYB Y` is now disabled unless `LOCKING` is also enabled, and except that the N (new-library) command now respects `READONLYB`, just like the U (update) command does.

`REPLROBR` *<Y/N/Q/W>*

The **ReplRoBr** parameter, "replace read-only browse," is used with the `READONLYB` parameter, to tell TLIB to let subsequent extracts replace read-only source files, (or, with the Q or W settings, read/write files which are not already checked-out to the current user ID).

The possible settings are:

```
 REPLROBR Yes (normal setting for network users)
 REPLROBR No  (the default)
 REPLROBR Querywritable
 REPLROBR Writable (dangerous!)
```

This parameter is useful if you are using check-in/check-out locking, and you have configured `READONLYB Y` to use the DOS read-only attribute to distinguish between browse-mode files and those which you have checked-out for modification. If `REPLROBR Y` is configured, then you can easily refresh the browse mode source files in your work directory, to ensure (for example) that you are compiling with the latest versions.

Note that a browse mode extract (EB) will not replace a file which is already checked-out for modification.

The default is N, do not allow replacement of read-only source files. Here we leave it set to the default.

```
 ReplRoBr N
```

Note: under certain circumstances configuring `REPLROBR Y` allows extracts to replace read-only files even when locking is disabled, notably when locking has been manually disabled via the C (configure) command.

The other two settings (Q or W) are special-purpose choices to allow replacement of read/write source files if they aren't checked-out/locked by the current user `ID`.

Normally, for silent replacement of browse-mode files, you would configure `READONLYB Y` and `REPLROBR Y`, and then TLIB's “`E`” (and “`EBF`”) commands will silently replace those files that have the read-only attribute set, and that are not checked-out to the current user `ID`.

“`REPLROBR Querywritable`” (or “`REPLROBR Q`”) is similar to `REPLROBR Y`, but if a source file that the user doesn't have checked-out/locked is writable, the user is asked whether or not he wants to replace a read/write source file. `READONLYB Y` must be configured to use this option. Read-only source files are silently replaced, just as when `REPLROBR Y` is configured.

“`REPLROBR Writable`” (or just “`REPLROBR W`”) lets TLIB replace writable browse-mode files without complaint, when `READONLYB N` is also configured.

If for some reason you don't want to configure `READONLYB Y`, you can configure `REPLROBR W` (writable), and TLIB's extract commands will silently replace files that you don't have checked-out, even if those files are not read-only.

(It is also possible to force silent replacement when extracting if you configure `REPLACE Y` or `REPLACE A`, but that is even more dangerous than `REPLROBR W`, and we strongly recommend against it.)

Beware: if you configure `REPLROBR W`, then TLIB's determination of whether or not it can silently replace a file is made solely on the basis of whether or not *you* have the file checked-out. Plus, if you have configured `LOCKING B` (per-project-level locking), then the determination only considers locks at the current project level. TLIB's check-in/out locking mechanism does not keep track of which computer or directory the files have been extracted into, so if you configure `REPLROBR W` and you use a different user `ID`, or if you've configured `LOCKING B` and have a file checked-out at a different project level, you may accidentally replace checked-out files! Therefor, if you configure `REPLROBR W`, it is *critically important* that you only use one user `ID` when working in a given work directory, and that you also use a different work directory for each project level if you configure `LOCKING B` (per-project-level locking).

This table relates the `REPLROBR` and `READONLYB` setting combinations that make sense together:

`REPLROBR` is	`READONLYB` must be	Silent replacement?
`N`	any	No
`Y`	`Y` or `W`	if read-only & not locked by you

`REPLROBR` is	`READONLYB` must be	Silent replacement?
`Q`	`Y` or `W`	if read-only & not locked by you
`W`	`N`	if not locked by you

For most users of multi-user editions of TLIB, `REPLROBR Y` and `READONLYB Y` are the best choices.

`SHEIGHT` *<12-66>*
`SWIDTH` *<80-132>*

The **SHeight** and **SWidth** parameters are used to specify the size of your screen. If the display screen on your computer is not 25 lines by 80 columns, then you may wish to set these two parameters.

The **SWidth** parameter tells command-line versions of TLIB how many lines to display at a time when showing you delta or version information. Note that the **LogWidth** parameter (p. 267) should usually be set to (at most) **SWidth** minus one.

The **SHeight** parameter tells TLIB how long a message can be without being split (wrapped) to occupy more than one line.

TLIB 5.0 extended these configuration parameters by adding the ability to determine the dimensions automatically/dynamically on most PCs. If you configure `SHEIGHT 0` (zero) and/or `SWIDTH 0`, then TLIB will interrogate DOS (or OS/2) for the current screen size and use the appropriate value(s).

This is not the default, however; the default is still 80x25.

If another program (e.g., an editor macro) needs to parse TLIB's output, you can configure `SWIDTH` to a very large number (up to 32765), to prevent TLIB from splitting (wrapping) its messages onto multiple lines. This makes it much easier for another program, which is reading TLIB's output, to tell where one message ends and the next begins.

These are the default settings:

```
 SHeight 25
 SWidth 80
```

`VALIDATE` *<Y/N>*

The **Validate** parameter can be used to relax a file naming rule which TLIB otherwise enforces. Normally, TLIB validates the extension of every file name which you specify, to ensure that it is a source file (and not a TLIB library file or lock file). If you specify a library file, it is an error.

So that it can tell the difference between source files and library files by examining their names, TLIB will not normally let you store a source file with the same 3-letter extension as the corresponding library file. For example, if you have `LIBEXT ?$?` configured, then you are prevented from using a source file name with a "`$`" as the second character of its extension.

For most users, this is helpful. However, a few customers need to give their library files and lock files the same names and extensions as their source files (but in different directories, of course!). To do this, you must disable the validation of file extensions, by configuring `VALIDATE N`. The default is:

```
Validate Y
```

Example:

```
VALIDATE N
LIBEXT ???????
LOKEXT ???????
PATH F:\LIBS\/F:\LOCKS\
```

`SLASHCONT` *<Y/N/M>*

For users who habitually enter multi-line comments when storing new versions of source files via the U or N command, the **SlashCont** parameter can be used to remove the necessity of putting a backslash at the end of each to-be-continued comment line. If you configure "`SLASHCONT M`" or "`SLASHCONT N`", then comments are terminated by a null comment line (i.e., just press `ENTER`).

Note that this makes an exception to the general rule that pressing `ENTER` at any prompt aborts the current operation: although just pressing `ENTER` at the first “`Comment line?`” prompt will still abort the operation, pressing `ENTER` at subsequent comment line prompts will cause the operation to complete and the library to be updated.

If you configure “`SLASHCONT No`” (or “`SLASHCONT N`”) then you will always be prompted for additional comment lines (with the last one to be followed by a blank line) regardless of whether or not you specified a comment on the DOS command line.

You can configure “`SLASHCONT Maybe`” (or just “`SLASHCONT M`”) if you don't want to have to enter a backslash at the end of to-be-continued comments, except for comments specified on the command line.

Like `SLASHCONT Y` (the default), this lets you run “`TLIB U`” or “`TLIB N`” with a comment on the DOS (or OS/2 or NT) command line, and not be prompted for additional comments for each file (unless the command-line comment ends in “\”).

However, like `SLASHCONT N`, this lets you interactively enter multi-line comments which are terminated by a blank line rather than needing backslashes at the ends of every line except the last one.

There are three cases:

1) *You specified a comment on the command line, with no backslash at the end.* You'll be prompted for additional comments only if `SLASHCONT N` is configured.

2) *You specified a comment on the command line, with a backslash at the end.* You'll be prompted for additional comments regardless of how `SLASHCONT` is configured. (But if `SLASHCONT N` is configured, the backslash becomes part of the first comment line.)

3) *You did not specify a comment on the command line.* You'll be prompted for additional comments regardless of how SLASHCONT is configured.

Similarly, there are three cases when entering additional comments:

1) *You entered a comment line with no backslash at the end.* If `SLASHCONT N` or `SLASHCONT M` is configured, then you'll be prompted for additional comment lines. But if `SLASHCONT Y` (the default) is configured, this will be taken as the last comment line, and the `U` or `N` command will finish.

2) *You entered a comment line with a backslash at the end.* You'll be prompted for additional comments regardless of how `SLASHCONT` is configured. (But if `SLASHCONT N` or `SLASHCONT M` is configured, the backslash will become part of the comment line.)

3) *You entered a blank line.* If `SLASHCONT Y` (the default) is configured, this aborts the update. If `SLASHCONT N` or `SLASHCONT M` is configured, then this aborts the update only if this is the first/only comment line; otherwise, it terminates comment-entry, and causes the `U` or `N` command to finish.

Summary:

	default		
SLASHCONT <Y/N/M>	Yes	No	Maybe
Requires "\" to continue interactive comments?	Yes	no	no
Allows mass wild-card updates w/o prompts?	Yes	no	yes

We suspect that most users will prefer `SLASHCONT Y` (the default, “yes”) or `SLASHCONT M` (“maybe”), rather than `SLASHCONT N` (“no”). Here we configure TLIB so that the backslashes need not be entered:

```
 SlashCont M
```

`AATTR` *<Set/Preserve/Reset>*

The **AAttr** parameter affects how TLIB sets the DOS “archive attribute” (a.k.a. “A” attribute) for your source files when you use the U or N command to store the source files into their TLIB libraries.

The “archive attribute” might better have been called the “needs-to-be-archived attribute.” It is a file attribute which DOS sets when a file is modified, so that BACKUP can tell that the file needs to be backed up. BACKUP resets the archive attribute after backing up the file.

Under DOS 3.2 or later, you can examine or change the archive attribute for one or more files by using the DOS “`attrib`” command; see your DOS manual.

The TLIB `AATTR` configuration parameter can be set three ways:

`AATTR SET` Causes your source file's archive attribute to usually be left alone, except that it will be set if TLIB modifies keyword or revision history information in the file because you'd configured `FIXKEYWD Y`.

`AATTR PRESERVE` Causes the source file's archive attribute to always be left alone, regardless of the `FIXKEYWD` configuration parameter.

`AATTR RESET` Causes the source file's archive attribute to always be cleared after a successful update of the library via a U or N command. The rationale for this mode is that storing your source code into the TLIB library is similar to backing up the source file with DOS's BACKUP command, so there is no need to back up the source file if its TLIB library file will also be backed up.

The default is:

```
 AATTR SET
```

`TOUCHSOUR` *<N/Y/M/R>*

The **TouchSour** ("touch source") parameter controls the file date with which a source file is left after an update (`U` or `N` command). (To “touch” the source file is to set its last-modified date/time stamp to “now.”)

There are four choices:

`TOUCHSOUR No`	Never touch the source file; preserve its date/time stamp even if TLIB modified it to update keywords or a revision history log. (This is the default.)
`TOUCHSOUR Yes`	Touch the source file if the library was successfully updated, regardless of whether or not TLIB modified the source file.
`TOUCHSOUR Modified`	Touch the source file only if TLIB modified it to update keywords or a revision history log (with `FIXKEYWD Y` configured), or to remove an embedded comment line (with `CMTFLAG` configured).

`TOUCHSOUR Revhist`	Touch source only if a revision history log was inserted (due to `FIXKEYWD Y`) or a comment line was removed (with `CMTFLAG` configured); i.e., if the line numbers changed. (This is useful for keeping source files "in synch" with debugger line numbers).

Note that this affects the date/time stored in the TLIB library file, as well as the date/time of the actual source file. However, it does not affect the date/time of any reference copy of the source file which TLIB creates due to the `r=y` option on the `LEVEL` configuration parameter; reference copies are always created with the current date/ time (which may be a few seconds newer than that of the source file in your work directory, even if `TOUCHSOUR Y` is configured).

Related parameters:

```
AATTR Set/Preserve/Reset  (p. 287)
FIXKEYWD Yes/No  (p. 277)
```

`CMTFLAG` *<number,1-80>*`,`*<quoted-string>*
`CMTSUFFIX` *<number,1-253>*`,`*<quoted-string>*

These two configuration parameters support TLIB's ability to take a comment from the top of the source file and use it as a supplemental TLIB comment line when you use the U (update) command to store a new version of your source file into the corresponding TLIB library file. This supplemental comment is added to the end of the regular TLIB comments. You must still enter the regular comments (either interactively, via the DOS command line, or from a comment file).

This feature is primarily intended to help support the M (“migrate changes”) command, which uses it to record DIFF3-merge histories in the TLIB comments. However, you can also use it for other purposes.

To use this feature, you must configure **CmtFlag**, which tells TLIB how to recognize the special comment line. The syntax for the `CMTFLAG` configuration parameter is similar to that of `LOGFLAG`:

```
CMTFLAG <first-column>,<quoted-string>
```

Example (C++):

```
cmtflag 1,"//CMT:"
```

This would tell TLIB to look for a C++ comment beginning with the string "`CMT:`". Thus, if you wanted to insert the supplemental comment, "`[MERGED_3.5_&_6,_base=3.3]`", you could put the following comment on the first line of your source file, starting in the leftmost column:

```
//CMT:[MERGED_3.5_+_6,_base=3.3]
```

If you wanted to indent the comment by two spaces, you could have configured:

```
cmtflag 3,"//CMT:"
```

Note that the first column is column 1; there is no column 0 the way that TLIB counts columns. Also, you may not use DOS redirection characters (`<>|`) in the quoted string, since it will appear on DIFF3 command lines in `MIGRATE2.BAT` after M (migrate) commands.

Some programming languages do not support comments that are terminated automatically at the end of the line. For these languages, TLIB provides the **CmtSuffix** configuration parameter, which works much like the `LOG-SUFFIX` configuration parameter:

```
CMTSUFFIX <minimum-starting-column>,<quoted-string>
```

Example (Pascal):

```
cmtflag 1,"{CMT:"
cmtsuffix 1,"}"
```

If you configure `CMTSUFFIX`, then TLIB will compare the end of your comment line with the configured string (normally a "close-comment" marker), and remove the close-comment marker from end of the supplemental comment. Thus, if you wanted to insert the supplemental comment "`[MERGED_3.5_&_6,_base=3.3]`", you could put the following comment on the first line of your source file, starting in the leftmost column:

```
{CMT:[MERGED_3.5_+_6,_base=3.3]}
```

Note that TLIB removes leading and trailing blanks from the supplemental comment before storing it, so you could also put the following comment on the first line of your source file, with exactly the same effect:

```
 {CMT: [MERGED_3.5_+_6,_base=3.3] }
```

The *<minimum-starting-column>* specifies a minimum column number for the suffix (close-comment); it should normally be configured to 1.

This version of TLIB only supports a single-line supplemental comment to be taken from the top of the source file. Future versions of TLIB may allow multiple lines.

Note #1: If you use this feature, then it is not advisable to configure `FIXKEYWD N`, since doing so will prevent TLIB from removing the then-obsolete supplemental comment line from your source file after updating the TLIB library with the new version.

Note #2: Although the `CMTFLAG` and `CMTSUFFIX` parameter names are (like all TLIB configuration parameter names) case-insensitive, the quoted strings are not. The strings must match exactly, or TLIB will not recognize them. Thus, “`//CMT:`” does not match “`//cmt:`”.

`CZTRUNC` *<Y/N>*

This rarely-used configuration parameter can be set to let TLIB truncate and store ASCII text files which contain ctrl-Z (end-of-file) characters.

CZTrunc affects what happens when TLIB is doing a U (update) command for a text file and it encounters a Ctrl-Z in the file which is not within 128 bytes of the end of the file. If `CZTRUNC N` is configured then TLIB will abort the update (this is the default). If `CZTRUNC Y` is configured, then TLIB will go ahead and update the library with the truncated file (after displaying a dire warning).

The default is:

```
 CZTRUNC N
```

`AUTOSET` <*file-name*>

The rarely-used **AutoSet** configuration parameter lets you change the path or name of the autoset file, or to disable the autoset facility altogether. (The autoset file, if it exists, is used by TLIB to define pseudo-environment variables; see p. 83 , for details.)

For instance, if you wanted `AUTOSET.BAT` (in the current directory) to be your autoset file in both DOS and OS/2 (rather than using `AUTOSET.CMD` under OS/2), then you could configure “`AUTOSET AUTOSET.BAT`”.

To disable the autoset facility, configure `AUTOSET` with no file name.

Note that the `AUTOSET` configuration parameter must be specified before any `%`*name*`%` references in your configuration file. After the first `%`*name*`%` reference, the `AUTOSET` configuration parameter is illegal.

Default is “`AUTOSET autoset.bat`” (under DOS), or “`AUTOSET autoset.cmd`” (in OS/2 protected mode).

`OLDNAME` <*Y/N*>

The **OldName** configuration parameter is obsolete.

`ONETHREAD` <*Y/N*>

The `ONETHREAD` parameter (formerly used for performance tuning under OS/2) is now obsolete.

`SLICKEPSI` <*Y/N/Maybe*>

The **SlickEpsi** configuration parameter is for users of editors which support a “concurrent process buffer,” such as MicroEdge's *SlickEdit* under

OS/2 and Windows-NT, and Lugaru's *Epsilon* editor under MS-DOS and OS/2.

Users of SlickEdit 2.2 or later, or of Epsilon, should not configure this, since TLIB can automatically adjust the default to the correct setting (by examining the `SLKRUNS` and `EPSRUNS` environment variables). Users of SlickEdit 2.1 for OS/2 should configure this to `SLICKEPSI Y`, or else set up SlickEdit to define the `EPSRUNS` environment variable, by defining the `SLICK` environment variable like this:

```
 set SLICK=-#"set EPSRUNS=1"
```

You may be wondering, “what is a concurrent process buffer?”

The “concurrent process buffer” provided by SlickEdit and Epsilon is a unique and wonderful feature. It provides a regular operating system command-line prompt *in* an editor window, even while you edit other files.

The compiler can be grinding away, finding more compile errors, even as the editor parses the error messages to find errors and let you fix them! You needn't wait for the build to complete before you start fixing the errors which the compiler finds.

Plus, since the input and output is being “logged” into an editor buffer, you can easily scroll back to find old error messages or commands, or use copy/cut/paste to edit and re-enter them. Very nifty.

However, there are some limitations. One is that the output from programs that “draw” on the screen simply doesn't look right. Things like carriage returns and backspace characters are discarded or displayed improperly. For instance, both SlickEdit and Epsilon break lines at line-feed characters in the concurrent process buffer, and ignore carriage-returns.

This is a problem for TLIB, since TLIB sometimes uses carriage-returns or backspace characters to make the display more attractive.

So, if you configure `SLICKEPSI Y`, then TLIB adjusts such messages to be more suitable for when TLIB is run in a concurrent process buffer.

Another problem is that (under some operating systems) the editor concurrent process buffers may not be able to run programs which accept character-at-a-time input. The Windows-NT version of SlickEdit has this problem. So, when `SLICKEPSI Y` is configured, TLIB does only line-at-a-time input from standard-input.

Note that running TLIB in an editor's concurrent process buffer puts you “in” your editor while entering TLIB comments (which is something we're planning to add to TLIB in a more conventional fashion eventually).

The default setting is `SLICKEDIT Maybe`, which means that the setting is automatically set according to whether or not TLIB determines that it is being run under SlickEdit or Epsilon.

`FILETYPE` *<Auto/Text/Binary/EOFtol/Runlen>*

The **FileType** configuration parameter selects between four possible library file formats.

The normal (text) format is especially well suited for storage of ASCII text files (such as program source code files). The “EOFtol” format is just like text format, except that the file can contain embedded ctrl-Z characters.

The two binary formats, “binary” and “runlen,” can store *any* type of file; use them for object module libraries, spreadsheet files, data base files, non-ASCII word processor files, etc..The difference between them is that “runlen” format uses runlength-compression to preprocess the files before generating the deltas. For sparse files it may substantially improve performance and reduce the size of the TLIB library files.

If FileType is configured to the default, “Auto,” then it examines the file before storing the first version, and selects an appropriate format automatically. However, you might want to use conditional configuration parameters (p. 321) to select which files are to be kept in “binary” or “runlen” format libraries. For example:

```
REM - most library files are text format
filetype auto
IF *.WK*
  REM - Lotus spreadsheets require binary-or runlen format
  filetype runlen
ENDIF
```

When binary or runlen format is selected, three other configuration parameters are ignored: `ENTABU`, `DETABE` and `ADDCTRLZ` are only meaningful for text files (except that AddCtrlZ will still affect the format of your journal file, if any).

You can store *any* kind of file in binary and runlen format libraries. They are highly useful for data base files, object module libraries, word processor files (for word processors which utilize non-text formats), etc.. Binary format works well with any file which goes through repeated revisions in which changes are fairly localized. However, TLIB's “revision history log” and “keyword” features, which allow version-specific information to be automatically updated in your source file, are not supported for binary format library files. Also, “delta review” does not work well (if you enter “`?`” at the “`Comment line?`” prompt to view a delta, you'll probably see gibberish).

If you program in dBase, Clipper, FoxBase, etc., you'll probably want to keep your data base structures under version control with TLIB (using `filetype binary`, or `filetype runlen` of course). To facilitate this, Mr. Michael Magen has kindly given us permission to distribute a program called `COPYSTEX`, which he wrote to extract the structure from `.DBF` files. It is rather large (because it is written in Clipper), so we have not included it with TLIB, but if you need a copy, please contact us, and we'll be happy to send it to you. (An alternative, if you have FoxDoc, nee SNAP, by Walter J. Kennamer, is to store the “data dictionary file” which it creates, instead of storing the individual data base structure files; SNAP or FoxDoc can re-create the empty `.DBF` structure files from the data dictionary file. Mr. Kennamer's address is 1801 E. 12th St., Apt. 1118, Cleveland, OH 44114.)

You can use binary format libraries for `.EXE` and `.OBJ` files, too. However, minor source code changes often cause wide-ranging object file changes in such files, so the calculated “deltas” will often be very large. When you add a new revision to a library of `.EXE` files, you can expect to the library to grow by 50-90% of the size of the `.EXE` file (unless the change is extremely minor).

TLIB's binary file support works like this: When you create a binary format library (with the N command), TLIB will do a statistical analysis of the file, trying to find a good set of “delimiter” bytes which will make it possible to break the file into variable-length “records” of manageable size. These delimiters are used by TLIB like carriage-return/line-feeds in a text file (which divide the text file into lines). The chosen set of delimiters is stored in the TLIB library. The analysis is rather time-consuming, and it

makes the N command run a bit slower for binary format libraries. However, the analysis is only done when a library is created, not when it is updated.
The "runlen" format is just like the "binary" filetype format, except that TLIB first does a simple "run-length" compression step before analyzing or storing the file. For "sparse" files, such as databases, this can greatly improve performance and storage efficiency.

The `FILETYPE` configuration parameter only affects the creation of new library files.

There is no way to directly change the format of an existing library file from binary to text or vice versa, but if you need to convert one or more TLIB libraries from FileType Binary to FileType Text, or vice-versa, you can use the TLIB-to-TLIB conversion utility, `TLIBTLIB.PL`, which can be found in the `CONVERT.ZIP` archive on your TLIB diskette (which also includes conversion utilities from various other version control systems to TLIB). `TLIBTLIB.PL` converts from one format to the other by running TLIB to extract each version from the old library and then store it in the new library. That is, it automates the tedious process of creating a new TLIB library containing all the versions and comments that were in the old one. It is slow, but it works. See the `CONERT.TXT` file for instructions.

Note that TLIB is not dependent upon the analysis phase deducing a "good" set of delimiters; however, a good set of delimiters will improve the efficiency with which "deltas" are calculated and stored.

Having determined the delimiters, TLIB can process a binary file as a sequence of variable-length records, much as it handles the variable-length lines in a text file, except that the library format is a bit different. Instead of "`.C`" and "`.I`" lines, it contains "`.C`", "`.J`" and "`.K`" records, which are stored in a "length byte + data" format rather than as carriage-return/linefeed delimited lines.

It is easy to tell whether a TLIB library is the (regular) text format or the (new) binary format. Binary TLIB libraries begin with "`.Vc`". Text format libraries begin with "`.Vt`" if blank-to-tab compression is not enabled. Text format libraries for which blank-to-tab compression is enabled (i.e., "`ENTABU Y`" was configured when the library was created) begin with either "`.V_`" or "`.V `". (See also p. 371.)

`TRACK` *<Y/N/Maybe>*

The **Track** parameter enables and disables version tracking; it is normally specified within an `IF/ENDIF` block (p. 321). Configure `TRACK Y` to enable version tracking, or `TRACK N` to disable it. Configure `TRACK MAYBE` (or just "`TRACK M`") to enable it only for those files that are already being tracked.

Suggestion: it is advisable to configure TLIB to track only certain files (like your source files). So, you should add something like the following to the TLIB configuration file (you can either add it manually or by running TLIBCONF):

```
TRACK N
IF *.C,*.H,*.ASM,*.BAT,MAKEFILE.*
  TRACK Y
ENDIF
```

If you configure `TRACK Maybe`, then TLIB will behave as if `TRACK Y` was configured for those files which are already listed in the current project level (or a predecessor level), but it will behave as if `TRACK N` was configured for those files not currently being tracked.

Also, if `TRACK Maybe` is configured, those modules already listed in the working directory tracking file but not in the project-level tracking file will continue to be tracked only in the working directory tracking file (as if `a=N` were configured in the `LEVEL` parameter for the current project level).

Note that the `TRACK` and `REFSUBDIR` parameters (unlike the `LEVEL` and `PROJLEV` parameters) can be specified within `IF/ENDIF` blocks, so that you can have different `TRACK` or `REFSUBDIR` settings for different files.

The default is:

```
TRACK N
```

See also p. 138.

`CREATETF` *<Y/N>*

The **CreateTF** parameter is used to tell TLIB to automatically create missing project-level tracking files ("createtf" is short for "create tracking file"). You can configure `CREATETF Y` if you would like TLIB to create project level tracking files automatically. However, you must still create the required reference directories manually.

The default is `CREATETF N`, which means that TLIB will only automatically create the working directory tracking file, not the project-level tracking files).

We anticipate that most programmers will want to leave this set to the default (`CREATETF N`), to avoid accidental creation of extraneous tracking files in the event that the `LEVEL`s are incorrectly configured. However, if you are the "system librarian" (the version control administrator / guru) at your company, you may wish to configure `CREATETF Y` to simplify setting up new project levels.

If `CREATETF Y` is not configured and TLIB fails to find a needed project-level tracking file, it displays a helpful error message which suggests configuring `CREATETF Y`.

The default is:

```
 CREATETF N
```

`AUTOBRNCH` *<Y/N/Q>*

The **AutoBrnch** parameter controls TLIB's automatic branch creation feature (p. 107), in which TLIB will automatically create a new branch when you update a library with a new version of a file, but (according to the record in the work directory tracking file) you didn't start with what is now the latest version.

Eariler versions of TLIB would just just go ahead and create the new branch version. Configure `AUTOBRNCH Y` to restore TLIB to this behavior.

If you want TLIB to ask you before creating a new branch version, then leave it configured to the default, `AUTOBRNCH Q`.

If you want TLIB to issue an error message and skip the file, then configure `AUTOBRNCH N`.

Note that this configuration parameter will not force everything to be stored as trunk versions. If that is the behavior you want, then you probably should just disable version tracking altogether, or else configure `PROJLEV *`.

The default is:

```
 AUTOBRNCH Q
```

`PROJLEV` *name*

The **ProjLev** configuration parameter is used to select the name of your current project level, which must be defined in a `LEVEL` configuration parameter unless you use one of the special pre-defined names, "`*`" or "=".

If you configure `PROJLEV =` (and `TRACK Y`), then TLIB's E (extract) command will consult the current work directory tracking file to determine which is the "current" version for extracting. That is, "`TLIB E`" will retrieve the version number indicated in the local `TLIBWORK.TRK`, which is the version you most recently stored or retrieved (but which is not necessarily the latest trunk version). This can be used to manage semi-custom software, but it is probably inappropriate unless you work alone, with locking disabled.

If you do not configure `PROJLEV`, or if you configure it with no *name* specified, then the TLIB E (extract) command will always retrieve the latest trunk version (rather than the version number in your work directory tracking file). This makes "`TLIB E file.ext`" equivalent to "`TLIB ES file.ext *`". This has the advantage of simplicity, and it is often the best choice for small projects, in which there is only one "current" level of code.

Configuring `PROJLEV *` is similar, except that it also disables TLIB's automatic branching feature. In other words, it makes "`TLIB U file.ext`" equivalent to "`TLIB US file.ext *`". See pp. 107 and 298).

For large projects with many programmers, it is usually better to use named project levels.

The default `PROJLEV` is none, which means that TLIB will always extract the latest trunk version (unless you specify a particular version number, e.g. via the ES command).

See also pp. 139 , 143 and 107.

Note: when your work directory is the reference directory for one of your configured project `LEVEL`s, then TLIB will detect that fact and automatically change or set your current `PROJLEV` to be the name of that `LEVEL`. See `WORKDIR` (p. 304).

`LEVEL n=`*name* `d=`*path* `p=`*name* `i=`*names* `s=`*{Old/New/Q/Changed}*
`a=`*{Y/N/Q}* `r=`*{Y/N}* `b=`*n* `f=`*{Y/N}* `w=`*{Y/N}*

The **Level** configuration parameter is used to tell TLIB about your named project levels. It is described in detail under “Configuring Your Project Levels,” p. 141.

Note: though the `LEVEL` parameter is shown here on two lines, it must be all on one line (of at most 254 characters) in your TLIB configuration file.

`TREEDIRS` *<Y/N>*

TreeDirs enables and disables tracking of “relative subdirectories.” Enable this by configuring `TREEDIRS Y` if you want TLIB to track a “tree” of related subdirectories as one logical unit.

There will be only one version tracking file for the entire “tree” of directories, and each “key” in the tracking file will contain a relative path along with the file name and extension for source files which reside in the “lower” subdirectories. See also `WORKDIR` and `DOTDOTOK` (below).

Restriction #1:

When `TREEDIRS Y` is configured, TLIB 5.50 also checks to be sure that your configured `LEVEL`s are do not have reference directories which are subdirectories of one another.

If you have configured `TREEDIRS Y`, you must not have configured the main reference directory for any of your configured `LEVEL`s be a subdirectory of the reference directory for any other configured `LEVEL`. Such a directory would simultaneously be a reference directory for two different `LEVEL`s at the same time, which TLIB does not allow. For example, the following combination is not allowed:

```
treedirs y
level n=abc d=f:\def\abc\
level n=def d=f:\def\
```

If you don't configure `TREEDIRS Y`, then this restriction does not apply.

Restriction #2:

Do not configure `TREEDIRS Y` in combination with `REFSUBDIR` (except for `REFSUBDIR nul`, see p. 166).

The default is:

```
TREEDIRS N
```

See also pp. 61 , 66 , 111 , 127 , 139 , 165 , 304 & 305.

`TOPRELATI` *<Y/N/Maybe>*

The TOPRELATI ("top relative") parameter affects the way that TLIB interprets `path\file` ("relative" or "unrooted") specifications under some circumstances, when you are working in a subdirectory other than the main work directory and `TREEDIRS Y` is configured.

The question that the TOPRELATI paramter answers is, how should `path\` be interpreted: is it relative to the main (top) work directory

(`WORKDIR`), or is it relative to the current subdirectory? The TOPRELATI ("top-relative") parameter enables you to tell TLIB how to interpret such file names.

Prior to TLIB 5.00m (circa August, 1993), it was assumed that all such paths were relative to the current directory. However, this prevented correct operation with file lists and snapshot files that included the relative subdirectories, since TLIB would erroneously interpret such paths as being relative to the current directory, rather than relative to the main work directory.

The three possible settings are:

`TOPRELATI Y` names are relative to WORKDIR

`TOPRELATI N` names are relative to current directory (like T LIB 5.00L)

`TOPRELATI Maybe` names are relative to current directory except when the
names are read from a TLIB snapshot (version label) file.
This is the default.

Note: The `TOPRELATI` parameter has no effect unless `TREEDIRS Y` is also configured `and the current directory is not WORKDIR.`

Sometimes you may need to build batch files in which the `TOPRELATI` parameter is adjusted for a single file. For greater convenience in such situations, TLIB also supports the `-r` command-line option, to force `TOPRELATI Y` for the remainder of the current command-line, only. The `-r` option should be specified before the command you wish it to affect. But if you use it in combination with the `-q` ("quiet") or `-d` ("debug") option, then specify the `-r` after the `-q` or `-d`.

You can specify `-r` or `-r1` to force `TOPRELATI Y`, when source file paths are relative to the main work directory (instead of relative to the current directory).

You can also specify `-r0` to force `TOPRELATI N`, if the source file paths are relative to the current directory rather than to the work directory.

Like the `TOPRELATI` configuration parameter, the `-r` option does not affect the operation of TLIB unless `TREEDIRS Y` is configured and the current directory is unequal to the work directory.

Note: The `-r` option is intended only for use with individual file names, not with wild-card specifications.

`DOTDOTOK` *<Y/N>*

If you configure `TREEDIRS Y`, TLIB 5.50 does some “sanity-checking” when it determines the “relative subdirectories” used in track file indices: the source file must be in your current work directory, or a subdirectory of it. If not, an error is reported.

There are two variants of the error:

a) the source file is not even on the same drive as the configured `WORKDIR`;

b) the source file is on the same drive, but not in `WORKDIR` or its subdirectories.

The first case (wrong drive) always generates an error.

However, the second case (right drive, wrong directory) will be tolerated by TLIB if you configure:

```
 DOTDOTOK Y
```

The `DOTDOTOK` configuration parameter simply determines whether TLIB will, if necessary, use double-dot pseudo-directories in the tracked relative directory specifications, thereby allowing source files to be anywhere on the drive.

Thus, for example, if your `WORKDIR` was `C:\WORK\` and you configured `DOTDOTOK Y`, then TLIB could track `C:\TEST\BIG.DOC` as `..\TEST\BIG.DOC`.

The default is `DOTDOTOK N` (double-dots are not okay). This is appropriate for most users.

Note #1: The `DOTDOTOK` parameter has no effect unless `TREEDIRS Y` is also configured.

Note #2: "`DOTDOTOK`" is pronounced "dot dot okay."

`WORKDIR` *path*

You can use the **WorkDir** parameter to tell TLIB which directory is your "working directory." Or, if you've also configured `TREEDIRS Y`, then `WORKDIR` can be used to tell TLIB which directory is the "root" of a "tree" of working directories.

Most users do not need to configure `WORKDIR`, since the default setting is usually adequate.

When `TREEDIRS N` is configured, the default for `WORKDIR` is simply the current directory ("`.\`").

However, if `TREEDIRS Y` is configured, then the default for `WORKDIR` is either the current directory or one of the "parent" directories. The algorithm for determining the default `WORKDIR` when `TREEDIRS Y` is configured is as follows:

1) First, TLIB examines the `LEVEL` configuration parameters (if any), looking for a project level which has as its reference directory either the current directory or one one of the current directory's parent directories. If one is found, then that directory becomes the `WORKDIR` (and the current `PROJLEV` setting is overridden).

Note: When you are working in the reference directory for a project level, and locking is enabled, you may *not* check-out/lock modules. You can extract for browse (EB command), but not for modification (E command).

2) Otherwise, TLIB examines the current directory and each of its "parent" directories, in turn, looking for a directory which contains a `TLIBWORK.TRK` file. (The "parent" directories are "`..\`", "`..\..\`", etc..) If the current directory does not contain a `TLIBWORK.TRK` file, but one of the parent directories does, then then the parent directory which contains `TLIBWORK.TRK` becomes the default `WORKDIR`.

3) If no `TLIBWORK.TRK` file is found, neither in the current directory nor in any of the parent directories, TLIB prompts the user to specify which directory is the work directory, and creates the `TLIBWORK.TRK` file there.

The question asked is similar to the following:

```
TREEDIRS Y was configured but the main/top work directory
could not be deduced, because TLIBWORK.TRK was not found in
or above the current directory, ‘C:\WORK\CURRENT\’.  Enter
the depth of the main work directory, 0-2, where 0 is ‘C:\’,
and 2 is ‘C:\WORK\CURRENT\’:
```

To prevent TLIB from asking this question the first time it is used in a new work directory, you can create an empty (0-3 byte long) tracking file in the work directory with any text editor, or with the DOS command:

```
ECHO.>TLIBWORK.TRK
```

`WORKDIR` can be set to a directory path that is at most 68 characters in length. Note, however, that the total path+name length for files is still limited to 80 characters, so having a very long WORKDIR will restrict your ability to use long file names under Win-95 (or, with `TLIB2.EXE`, under OS/2 or NT).

Note: If you configure `QUERIES N` (to prevent TLIB from asking questions of the user), and TLIB cannot determine the main work directory, then TLIB aborts rather than allowing the user to specify the work directory.

`REFSUBDIR` *directory-name*

The **RefSubdir** configuration parameter can be used in combination with an `IF`/`ENDIF` block if you are not using `TREEDIRS Y` but you nevertheless need to keep the reference copies of your include files in a different directory from the reference copies of your main source files.

See “reference directories” (p. 164) for details, including why you may need this.

There is no default for the REFSUBDIR parameter.

`FORCEREFR` *<Y/N>*

The **ForceRefR** (force reference copy refresh) configuration parameter affects what TLIB does if you've configured your `LEVEL` parameter to keep the reference directory up-to-date (`r=Y`), and you do a U (update) command which does not store a new version because there were no changes.

By default (`FORCEREFR N`), TLIB will not create a reference copy of the source file in the reference directory (because it didn't change the TLIB library). If you would prefer that TLIB go ahead and create the reference copy, you can configure `FORCEREFR Y`.

The default is `FORCEREFR N`.

See “reference directories” (p. 164) for more information.

`FIND1FILE` *<Y/N>*

The **Find1file** configuration parameter selects one of two behaviors when you specify a single, specific source file to TLIB (as opposed to a wild-card specification). If you configure:

```
 FIND1FILE Y
```

then TLIB will handle even exact file names as if they were wild-card specifications (unless the N wild-card search mode suffix is added to the command).

But if you configure:

```
 FIND1FILE N
```

then TLIB will handle fully-specified file names (non-wild-card names) without any wild-card searching.

This means (for instance) that if `FIND1FILE Y` is configured, then when you are using the project-oriented search modes (`A` and `T`) you cannot extract a file that is not listed in the project level(s), since TLIB will be unable to find the file. Since in project-oriented mode, the `A` search mode is the default for the `E` (extract) command (i.e., the `E` command is equivalent to `EA`), to extract a file which is not listed in the project level(s) you would need to override the wild-card search mode (i.e., use `EL` instead of `E`).

For an example of why you might want to configure `FIND1FILE Y`, suppose you are using tree-structured work directories (`TREEDIRS Y`) and you have several different files called `makefile` in various subdirectories. If you configure `FIND1FILE Y` and do the command:

```
 TLIB EI MAKEFILE
```

then TLIB will extract all the `MAKEFILE` files (into the appropriate subdirectories). But if `FIND1FILE N` is configured, that command will only extract the copy of `MAKEFILE` that belongs in the current directory.

`RELAXVERS` *<Y/N>*

The **RelaxVers** (relax version number restriction) configuration parameter

If you configure `RELAXVERS Y`, then TLIB will allow you to create versions with branch or trunk number zero, and/or to skip versions. This is not intended for general use, but rather to allow conversion of PVCS and RCS files to TLIB, via the `PVCSTLIB.EXE`, `RCS2TLIB.EXE`, and `GNU2TLIB.EXE` conversion tools (in `CONVERT1.ZIP` or `CONVERT2.ZIP`, on the TLIB distribution diskette).

Note #1: Support for version number zero and skipped version numbers was new to TLIB 5.01. If you use skipped or zero version numbers, then your TLIB libraries will `not be compatible with TLIB 5.00m and earlier.`

Note #2: To specify a zero branch version number, you `must always also specify the parenthesized number-of-the-branch.`

For example, suppose that you had a PVCS archive from which you built an equivalent TLIB library, using `PVCSTLIB.EXE` (or `PVCSTLIB.AWK`). Suppose, also, that the PVCS archive contained PVCS branch version number "3.2.1.0". Then the equivalent TLIB 5.50 version number would be "3:2.(1)0", and you `cannot abbreviate it to "3:2.0".`

This is a special case which applies only to branch version zero of branch number one, ".(1)0". For any other branch version within branch number one, the "(1)" can be omitted for brevity, so that, for example, "3:2.(1)4" is equivalent to "3:2.4".

This restriction exists is so that we can maintain compatible behavior with earlier TLIBs, which considered, for instance, "4.0" to be another way of referring to version "4".

The default is `RELAXVERS N`.

`TRACKEXT` *extension*

This rarely-used configuration parameter lets you to change the extension of the `TLIBWORK.TRK` version tracking file. Configure this only if you use the default file extension “`TRK`” for some other purpose.

The default is:

```
 TRACKEXT TRK
```

`ELSEWHERE` *<Y/N>*

TLIB 5.0 added a new configuration option, **Elsewhere**, which subtly affects the operation of the EBF (refresh browsed files) command.

Normally, the EBF command will extract any named source file which does not already exist in the work directory, as well as those which are determined to be out-of-date according to the information in the work directory's `TLIBWORK.TRK` file.

However, one of our users devised a scheme for taking his work home, in which the source files are not left in the working directory, and he needed TLIB to ignore the absence of the source files, and make its determination of which files to extract solely on the basis of the information in `TLIBWORK.TRK`.

To tell TLIB that his source files are elsewhere, he configures:

```
ELSEWHERE Y
```

However, most users should leave this configuration parameter set to the default, `ELSEWHERE N`.

`FNAMECASE` *<U/L/A>*

The `FNAMECASE` configuration parameter controls the “case” (upper-case vs. lower-case) of file names. The 3 choices are:

```
FNAMECASE Upper    - Force all file names to upper-case
FNAMECASE Lower    - Force all file names to lower-case
FNAMECASE Auto     - Behavior depends upon operating system
```

This affects the case of file names recorded in the journal file, and the “`% n`” keyword, as well as the actual names used when creating files.

The default is `FNAMECASE A`.

If you have a case-sensitive network server (e.g., a Unix machine), you may want to configure `FNAMECASE L`.

See also `LONGNAMES`, p. 77.

`SAY` *message*
`WARN` *message*
`ABORT` *message*

The `SAY`, `WARN` and `ABORT` parameters support generation of custom error and warning messages.
Use the `SAY` parameter to display a message to the console (in the Windows version of TLIB, the message goes in the Status Log). For example:

```
say Please don't keep modules checked-out/locked for weeks!
```

`WARN` is similar to `SAY`, except that a TLIB "`ERROR: in configuration file`" message will also be displayed.

`ABORT` is just like `WARN`, except that TLIB will halt after displaying the error message, rather than continuing.

Note that if you make your tlib.cfg `SAY` a message that begins with "`Note: `", "`Warning: `", or "`ERROR: `" then command-line versions of TLIB will colorize it and/or prevent it from scrolling off the screen (according to how the `COLORIZE` and `ERRORPAUS` parameters are configured), and Windows versions of TLIB will pop up the message in a dialogue box for the user (as well as putting it in the status log).

For example:

```
 say Note: Please don't lock files for weeks at a time!
```

These parameters are most often used in `IFF`/`ENDIF` blocks, to warn about error conditions.

`WORKDEPTH` *nn*

The `WORKDEPTH` configuration parameter can be used to specify the minimum subdirectory depth for a work directory. For example:

```
 WORKDEPTH 0   (the default: work directories can be in or
 below the root directory)
 WORKDEPTH 1   (work directories must be at least 1 level bel
ow
 the root directory)
 WORKDEPTH 2   (work directories must be at least 2 levels be
low
 the root directory)
```

This is useful, for example, to prevent the accidental use of root directories as TLIB work directories, which is important when `TREEDIRS Y` is configured (p. 300).

`NEWLINE` *<CRLF/LF/CR>*

TLIB transparently handles DOS, Unix, and Mac-format ASCII text files. As input to TLIB (for updates), any of the three formats are now handled equivalently. For output (extracts), the default is DOS format (CR+LF), but you can control this with the `NEWLINE` configuration parameter:

`NEWLINE CRLF`	The default, DOS format (carriage return + line feed)
`NEWLINE CR`	Mac format (carriage returns only)
`NEWLINE LF`	Unix format (line feeds only)

Note: this only affects the operation of TLIB in `FILETYPE TEXT` mode. If your TLIB library is in `FILETYPE BINARY` format, then the `NEWLINE` configuration parameter is ignored, and no end-of-line translations are done. Also, this only affects the handling of your text files; TLIB's library files are still stored in DOS format (with CR/LF after each line).

There is one small side-effect to this new feature that may affect some users of older versions of TLIB. TLIB now handles CR/LF, LF alone, or CR alone in your text file as all being equivalent. Earlier versions of TLIB handled CR/LF or CR alone as being equivalent, but LF alone was handled as a plain text character.

So, if you had a file that contained a spurious line-feed character somewhere, you may notice this difference in behavior, because if you extract the file (`TLIB E`) and then immediately update (`TLIB U`) TLIB will report that the file has changed! What happened is that the line containing the lone LF character was seen as a single line in the TLIB library and during the extract, but was split into two lines during the update. If you let TLIB store the new "changed" file, and then extract it again, you will find that your text file now has a CR+LF (or whatever you configured for the `NEWLINE` parameter) in place of the LF that it had there before.

Caveat: Ctrl-Z is still recognized as an end-of-file character, even in Unix-format files. Ctrl-D is not recognized as special in any way. So, if you have a file that contains a ctrl-Z character and you don't want TLIB to truncate the file at that point, you must configure `FILETYPE BINARY` for that file when you create the TLIB library for it.

`ERRORPAUS` *<0-3>*

The `ERRORPAUS` parameter controls whether or not TLIB will pause if an error or warning message has been displayed. Pausing after errors or warnings is intended to prevent important messages from being overlooked because they scrolled off the screen before being read.

You may configure `ERRORPAUS` to one of these four settings:

`ERRORPAUS 0`	(disable pauses after error & warning messages)
`ERRORPAUS 1`	(pause if "`ERROR`" was displayed, but not for "`Warning`" or "`Note`"; this is the default)
`ERRORPAUS 2`	(pause if "`ERROR`" or "`Warning`" was displayed, but not for "`Note`")
`ERRORPAUS 3`	(pause for "`ERROR`", "`Warning`" or "`Note`")

If you want to start a long job and return later to see whether there were any errors, then you may prefer to configure `ERRORPAUS 0` and simply redirect output into a file, for later inspection. TLIB will detect the fact that output has been redirected, and error and warning messages will be written to both "stderr" (usually the console) and "stdout" (the file to which you have redirected output). If, when you return, you see error messages on the screen, you can inspect the file with your redirected TLIB output.

Note that the `ERRORPAUS` parameter only affects operation in "non-interactive mode" (i.e., when all TLIB commands and parameters were specified on the command line).

If you don't specify the TLIB commands and parameters on the command line, then TLIB will operate in "interactive mode," and prompt you for them. In this mode, TLIB always pauses after each screen of text, even if there are no errors, and regardless of the `ERRORPAUS` setting.
The default is `ERRORPAUS 2`, which pauses for "`ERROR`" and "`Warning`" messages, unless you configure `QUERIES N` but leave `ERRORPAUS` unconfigured. In that case, the default is `ERRORPAUS 0` (pauses disabled). (This exception is to avoid breaking some front-end programs and editor macros which configure TLIB with `QUERIES N` because they depend upon TLIB never prompting for user input.)

See also: `EXITPAUSE` (p. 313) and `COLORIZE` (p. 313).

EXITPAUSE <*Y/N*>

The EXITPAUSE parameter can be used to make TLIB pause before exiting, even if no errors have occurred. If you want TLIB to always pause before exiting, you can configure:

```
 EXITPAUSE Y
```

The default is:

```
 EXITPAUSE N
```

See also: ERRORPAUS, p. 312.

COLORIZE <*Y/N*>

To help you avoid overlooking error and warning messages, command-line versions of TLIB can "colorize" them using ANSI excape sequences. You can control this feature with the COLORIZE configuration parameter. By default, if ANSI.SYS support is available, TLIB will now attempt to highlight error and warning messages through the use of ANSI escape seqences to selecting the colors. To disable this, configure:

```
 COLORIZE N
```

To force TLIB to colorize error and warning messages, without testing whether ANSI.SYS support is available, you can configure:

```
 COLORIZE Y
```

The default is to check for ANSI.SYS support, and colorize only if ANSI.SYS is loaded; this can be explicitly configured as:

```
COLORIZE Maybe
```

Note #1: TLIB also suppresses the ANSI escape sequences when in `SLICKEPSI` mode; that is, when running in the "concurrent process buffer" of the SlickEdit and Epsilon editors. So, an alternate way to prevent colorization is to configure `SLICKEPSI Y`. This will avoid warnings with older versions of TLIB that don't support the new `COLORIZE` configuration parameter. However, this also affects some other aspects of console I/O, since the SlickEdit and Epsilon concurrent process buffers cannot support some operations (such as single-character-at-a-time keyboard input).

Note #2: if you want a colored DOS prompt, but TLIB's ANSI escape sequences interfere with it, there are three things you can do to solve the problem:

a) You can simply disable TLIB's ANSI escape sequences, by configuring `COLORIZE N`. Unfortunately, this will prevent colorization of TLIB's error and warning messages.

b) To have colorized TLIB error and warning messages, and also retain your colorful DOS prompt, you can set your DOS prompt environment variable to select the prompt color of your choice, and use TLIB's `BANNER` configuration parameter to reset the screen color to white when TLIB starts up.

Thus, your `PROMPT` setting in `AUTOEXEC.BAT` (and in `CONFIG.SYS`, under OS/2) might be, for example:

```
Rem - bright, light blue prompt
PROMPT=$e[1;36m$d $t $p]
```

To your `TLIB.CFG`, add:

```
Rem - reset screen color to normal (white) at TLIB start-up:
numbanner 1
banner 1,"←[0m"
Rem - (where "←" is the escape character, ASCII 27)
```

c) You can use the `COLOROFF` configuration parameter to reset your screen colors to whatever you wish following display of the colorized word ("ERROR:" or "Warning:" or "Note:"). See `COLOROFF`, below.

See also: `ERRORPAUS`, p. 312.

`COLOROFF` *<string>*

The `COLOROFF` configuration parameter lets you tell TLIB what ANSI escape sequence to use to resets your console color to "normal" after displaying a colorized error or warning message.

This configuration parameter does nothing in TLIB for Windows, and does nothing if `COLORIZE N` is configured (which is the default under DOS if `ANSI.SYS` is not loaded).

The default is "`COLOROFF ←[0m`", where "←" is the esc character, ASCII 27.

`MAKEDIRS` *<Y/N>*

The `MAKEDIRS` configuration parameter is used to tell TLIB to automatically create missing directories, as necessary, in most circumstances.

To tell TLIB to create directories, configure:

```
 MAKEDIRS Maybe
```

or

```
 MAKEDIRS M
```

A few networks and operating systems may have bugs which cause incorrect error codes to be returned under various circumstances, such as when a file cannot be created because the directory is missing. If `MAKEDIRS M` does not work for you, and you suspect that this might be the reason, then you can try configuring:

```
 MAKEDIRS Y
```

Configuring `MAKEDIRS Y` causes TLIB to assume that a missing directory could be the true cause for a file-create failure, even if an error code was returned indicating some other cause. Do not configure `MAKEDIRS Y` unless you suspect that your network or operating system has this bug.

The default behavior for TLIB is not to create missing directories, but to give an error message, instead. This is similar to what happens if you configure:

```
 MAKEDIRS N
```

The only difference between configuring `MAKEDIRS N` and leaving the `MAKEDIRS` parameter unconfigured is that if `MAKEDIRS` is not configured then TLIB can also display a "hint" when a file create fails due to a missing directory. The hint suggests that configuring `MAKEDIRS M` might solve the problem.

`NT351BUG` *<Y/N>*

The `NT351BUG` configuration parameter enables TLIB to work around OS and network bugs which cause file sizes to be incorrectly reported for directory look-up ("findfirst") operations. This bug is characteristic of Windows-NT 3.51, but has been occasionally seen in other environments, as well.

When an application either closes a file or does a "file commit," the operating system is supposed to update the directory information to reflect an changes in file size and date. However, Windows-NT 3.51 has a bug which causes directory information to be incorrectly reported for a varying period of time (typically 1-15 seconds) after the file close or file-commit operation.

We paid Microsoft $150 for the privilage of reporting this bug, and their response, believe it or not, was that they had disabled file-commits for 16-bit applications, as an "optimization," and they did not intend to fix it in any NT 3.51 service pack. Fortunately, it seems to be fixed in Windows NT 4.0.

(Please, don't anybody tell them that they could optimize NT even more, and make it go even faster, if they disabled writes, too.)

So, we built a workaround into TLIB, to use an open/lseek/close sequence to look up file lengths, rather than the normal (and much faster) findfirst (directory look-up) approach.

If you need to configure this option, TLIB will tell you so. Otherwise, don't, since it adds significant overhead to some operations. (This is in contrast to the `NTFS35BUG` parameter, which adds very little overhead.)

To work around the bug in Windows-NT 3.51, and also a bug that was present in NT 3.5, configure these two parameters:

```
NT351BUG Y
NTFS35BUG Y
```

The defaults are `NT351BUG N` and `NTFS35BUG N`.

`NTFS35BUG` *<Y/N>*

The `NTFS35BUG` configuration option is used to tell TLIB to work around a bug in Windows-NT 3.5's NTFS file system. If you are running Windows NT with an NTFS file system, either on your workstation PC or your network file server, you should configure:

```
NTFS35BUG Y
```

This tells TLIB to do an explicit file-commit after appending additional data to a file, such as a TLIB library. If this is not done, then directory lookups (find-first) sometimes report the wrong file sizes for recently-closed files.

Note: do not configure both `NTFS35BUG Y` and `USEDUPHAN Y`.

See also: `NT351BUG`, p. 316.

`SHOWLNAME` *<Y/N>*

The `SHOWLNAME` (SHOW Library NAME) configuration parameter can be used to suppress TLIB's routine display of TLIB library file paths and names. For more concise messages, you can configure:

```
 SHOWLNAME N
```

This inhibits display of library & lock file names in most common TLIB messages. The default is still `SHOWLNAME Y`. (See also the `QUIET Y` configuration parameter.)

`LONGNAMES` *<Y/N/M>*

The `LONGNAMES` configuration parameter can be used to prevent TLIB for Windows and TLIBX from handling long file names under Windows 95, or to prevent TLIB2 from handling long file names under OS/2 and Windows-NT. See p. 77.

`SERIALNO` *<serialnumber>*

The `SERIALNO` configuration parameter is how you tell TLIB what what its serial number(s) are.

For example, suppose 504-01234-5-123456789012 was your TLIB serial number. (This serial number means that you have TLIB 5.50, yours was the 1234-th TLIB sold, and you bought a 5-user license; "123456789012" encodes the type of license you have, and a checksum.) You would configure:

```
 serialno 504-01234-5-123456789012
```

From then on, TLIB's copyright banner would say something like:

```
 TLIB 5.50g, 5 user license. Copyright 1985-1996 Burton Syste
ms Software
```

If you have more than one serial number, just configure them repeatedly, like this:

```
 serialno 500-01233-1-123456789012
 serialno 500-01234-5-234567890123
```

TLIB will sum the number of user licenses:

```
 TLIB 5.50g, 6 user license. Copyright 1985-1996 Burton Syste
ms Software
```

The `SERIALNO` parameter is usually used in the `TLIB.SER` file rather than the regular `TLIB.CFG` configuration file. `TLIB.SER` is an optional TLIB configuration file which, if it exists, must reside in the same directory as the TLIB executable (`TLIB.EXE`, `TLIBX.EXE`, `TLIB2.EXE`, or `TLIBDL-L.DLL`). `TLIB.SER` is read before the regular `TLIB.CFG` file. `TLIB.SER` is primarily intended to contain your TLIB serial number(s).

`USEUMBS` *<Y/N>*

The `USEUMBS` configuration parameter controls whether or not the real-mode DOS version of TLIB will attempt to "link" and use "UMBs" (Upper Memory Blocks), if available, under DOS 5.0 and later. Upper Memory Blocks are conventional memory areas between 640K and 1024K, which are commonly provided on 80386 or better computers by programs such as EMM386, 386MAX and QEMM.

There are two possible settings:

`USEUMBS Y`	-	TLIB will link Upper Memory Blocks if they are available. This is the default.
`USEUMBS N`	-	TLIB will not link upper memory blocks.

Note: Our CTMAP memory manager can also provide upper memory blocks on some 80286 and 80386 computers which use Chips & Technologies chipsets; see `CTMAP09C.ZIP`. However, CTMAP's UMBs are always "linked" and available, so they are unaffected by the TLIB “`USE-UMBS`” configuration parameter.

The `USEUMBS` parameter is ignored in protected mode: by TLIB2 under OS/2 or Windows-NT, and by TLIBX (the DOS Extended TLIB) when DPMI, VCPI or XMS extended memory is available.

By default, if TLIB runs low on memory under DOS 5.0 or later, it will ask DOS to “link” any available upper memory blocks, thereby making them available as normal DOS memory to programs like TLIB. When TLIB does this, it displays a message:

```
*** UMBs linked ***
```

For most users, this is good: it means that TLIB has additional memory that it can use when processing large files.

However, if you have a problem with badly interacting programs, that you suspect might be a memory usage conflict, especially when running a DOS “task switcher” or similar program, you may wish to configure:

```
UseUMBs N
```

This will prevent TLIB from attempting to use Upper Memory Blocks. (You can also accomplish the same thing by using DOS's SETVER command to make TLIB think it is running under DOS 4.)

`READONLYT` *<Y/N/W>*

The `READONLYT` configuration parameter controls whether or not TLIB will set tracking files (`tlibwork.trk`) to read-only. There are three possible settings:

`READONLYT Yes`	-	TLIB sets all tracking files to read-only when not open. This is the default.
`READONLYT No`	-	TLIB does not set tracking files to read-only.
`READONLYT Workdir`	-	Only work directory tracking files will be marked read-only. Project level tracking files will be left read-write.

This parameter is intended for use with multi-user editions of TLIB when used with Sun PC-NFS, since apparently with PC-NFS only the “owner” of a file can change the file from read-only to read-write.

This parameter can also be used to work around a bug in some versions of Novell Netware. This Netware bug causes files to lose their “sharable” attribute (which you set with Novell's `FILER` utility) whenever a program (such as TLIB or DOS's `ATTRIB` command) changes the read-only attribute. See `READ_ME.TOO` for details.

Conditional configuration parameters

`IF` *<list of wild-card specs>*
`ENDIF`

TLIB configuration files can contain two kinds of conditional configuration constructs: run-time conditionals and load-time conditionals. Run-time conditionals are in `IF/ENDIF` blocks (note: there is no `ELSE` clause). Load-time conditionals are in `IFF/ELSE/ENDIF` blocks. Run-time conditionals are used to specify configuration parameters which only apply to certain files, which are selected by wild-cards (sorry, you cannot use file lists in this context). Load-time conditionals are similar to "conditional compilation" for configuration files; they are predicated upon conditions that can be tested as TLIB is reading its configuration files, at program start-up (or, in TLIB for Windows, when the current directory is changed).

The syntax for run-time conditionals is:

```
IF <list of wild-card specs>
  REM conditional parameters go here
ENDIF
```

A few of TLIB's configuration parameters cannot appear in an `IF/ENDIF` block; TLIB will give you a clear warning if you try a prohibited one.

Example:

```
REM  These are popular changes to the “default” parameters:
EqualDate Y
```

```
path c:\tlibs\
libext ?$?????
lokext ?^?????
REM  No tab/blank conversions except on Pascal source code:
IF *.PAS
  entabU Y
ENDIF
REM  Revision history log insertion only for C modules:
IF *.C,*.H
  logwidth 76
  logflag 3,"--=>revision history<=--"
  logprefix 1,"/*"
  logsuffix 78,"*/"
ENDIF
detabE M
```

Parameters are processed in the order you specify them: if a single parameter is set more than once, only the last one has any effect.

`IF` blocks can be nested as deeply as you wish. Nested `IF` blocks can be used to make "exceptions" for particular files or groups of files. For example:

```
if *.wk?
  filetype binary
  if timsfile.wkl
    filetype text
  endif
endif
```

Note that only TLIB itself (`TLIB.EXE`, `TLIBX.EXE`, `TLIB2.EXE`, and TLIB for Windows) recognizes `IF/ENDIF` constructs; the DIFF3 and CMPR simply ignore everything between `IF` and `ENDIF`. You can use this feature to configure parameters for TLIB which do not affect CMPR or DIFF3, as in the following example:

```
if *.*
  addctrlz Y
endif
```

Note: TLIB 5.0 extended the syntax for the `IF` construct in configuration files, to allow multiple wild-card specifications, separated by commas. This feature may let you significantly reduce the size of your TLIB configuration file. For example:

Old way (still works, but now needlessly verbose):

```
IF *.DBF
  filetype binary
ENDIF
IF *.WK*
  filetype binary
ENDIF
IF *.EXE
  filetype binary
ENDIF
```

New way (much more concise):

```
IF *.DBF,*.WK*,*.EXE
  filetype binary
ENDIF
```

Note: you must *not* put any whitespace either before or after the commas! Thus, the following will *not* work:

```
IF *.DBF, *.WK*, *.EXE
  !does not work!
  filetype binary
ENDIF
```

`IFF` *<expression>*
`ELSE`
`ENDIF`

TLIB supports very flexible load-time conditionals in its configuration files. The following configuration directives are provided:

```
IFF <expression>
ENDIF

IFF <expression>
ELSE
ENDIF
```

Note that the `IFF`/`ENDIF` and `IFF`/`ELSE`/`ENDIF` directives are quite different from the `IF`/`ENDIF` construct. `IF`/`ENDIF` is used to specify configuration parameters which pertain to files that match one or more wild-card specs. An `IF` parameter's wild-card specification is re-tested for each file that TLIB processes.

In contrast, the `IFF`/`ENDIF` and `IFF`/`ELSE`/`ENDIF` directives are processed only as `TLIB.CFG` is being read. (For command-line versions of TLIB, that means they are processed only when TLIB loads; however, TLIB for Windows re-reads `TLIB.CFG` every time you change the current directory.)

In addition, `IFF` tests are conditioned upon Boolean expressions, rather than upon wild-card specifications. Boolean expressions are expressions of the sort used in a `LET` assignment, but with the added constraint that the result must be numeric. Zero is taken as "false" and non-zero means "true". (For a full description of the expression syntax, see the `LET` parameter, p. 270.)

Example:

```
 iff '%NECESSARY%'==”
 warn Oh, dear!  Environment variable %%NECESSARY%% is undefi
ned!
 endif
```

Example:

```
 let ONE=5-4
 iff %ONE%==1
   say Hey, it works!
 else
   abort This is impossible!
 endif
```

Example:

```
 REM  This example displays the current directory's path.
 REM  First, get "truename" of current directory:
 Let currentdir=UNQ NAM ".\"
 REM  It may not have a trailing slash, so add one:
 Let lastch= '%currentdir%' SST '-2:1'
 iff ('%lastch%' NE '\') AND ('%lastch%' NE ':')
   Let currentdir = UNQ ('%currentdir%' . '\')
 endif
 REM  Display the result (and also workdir)
 Say currentdir=%currentdir%, workdir=%tlibcfg:workdir%
```

Useful example:

```
 REM  Include "personal.cfg" from current or parent folder,
 REM  0-3 levels up.  1st, find the personal.cfg.  (If file
 REM  does not exist, SIZ operator returns -1 for its size.)
```

```
set pname='personal.cfg'
iff (SIZ %pname%) == -1
 let pname='..\' . %pname%
 iff (SIZ %pname%) == -1
  let pname='..\' . %pname%
  iff (SIZ %pname%) == -1
   let pname='..\' . %pname%
   iff (SIZ %pname%) == -1
    set pname=”
   endif
  endif
 endif
endif
iff ” eq %pname%
 say Warning: personal.cfg not found, default config used.
else
 let psize=siz %pname%
 let fullpname=uc unq nam %pname%
 let pname=unq %pname%
 include %pname%
 say Note: %psize%-byte %pname% (path %fullpname%) loaded.
endif
```

Or, for something really obscure:

```
Rem  Set %X% to any integer value:
let x=-46
Rem  Then test the MOD operator:
let easyway= %x% mod 3
let hardway= %x% - (3*(%x% / 3))
iff %hardway% <> %easyway%
 abort This is impossible!
else
 say (%x% MOD 3) = %EasyWay%   (remainder left after %x%/3)
endif
```

For a more realistic example, involving the management of many semi-custom software project levels, see p. 161.

See also “environment variables...” (p. 80), and the `LET` parameter (p. 270).

Compressed (archived) library files

`ARCCMD` *<path-of-pkpak.exe>*

`UNARCCMD` *<path-of-pkunpak.exe>*
`ARCTEMP` *<temporary-directory>*
`ARCEXT` *<extension>*

Note: use of this feature is deprecated.

Note to users of TLIB 4.12: This works just as it did in TLIB 4.12, except that nearly everyone now uses PKZIP & PKUNZIP instead of PKPAK and PKUNPAK.

(In fact, we hesitate to recommend using archived library files at all. Using them makes TLIB sluggish and terribly verbose, and it is not safe in a networked/multiple-programmer environment. We tried to remove this feature from TLIB 5.0, but too many of our beta-testers complained, so we relented and put it back in.)

TLIB can also modestly compress its library files, but only for binary files (“`filetype runlen`”), and it keeps the library file small by calculating deltas in a highly efficient manner and optionally converting multiple blanks to tabs. However, the DOS version of TLIB does have the ability to invoke PKZIP to automatically compress and decompress library and lock files, storing them in compressed archive format. PKZIP usually compresses text format TLIB library files by 50-60%, and you can store multiple files in a single archive to reduce the amount of space wasted by DOS's cluster-at-a-time disk allocation.

This can save disk space, but it has several disadvantages. It slows down the operation of TLIB considerably. Also, you may find the interspersed PKZIP messages distracting. Plus, we cannot guarantee compatibility between TLIB and future versions of PKZIP. So, unless you are very short of disk space, you will probably not want to use “compressed/archived libraries” with TLIB.

Important note #1: Even if there is a copy of PKZIP and/or PKUNZIP included with TLIB, the price of TLIB does *not* include a license to use PKZIP (except to unpack the TLIB diskettes). Licenses for PKZIP should be obtained directly from PKware, Inc., 9025 N. Deerwood Drive, Brown Deer, WI 53223. Telephone: (414) 354-8699. Fax: 414-354-8559.

Important note #2: The use of compressed/archived library files is *not recommended* with the network version of TLIB. The problem is that if two users try to access the same library or lock file at exactly the same time, TLIB's network synchronization mechanisms will not work properly, since TLIB will be operating upon temporary copies of the library and lock files.

Important note #3: TLIB does not “understand” the format of `.ZIP` files. Consequently, some forms of wild-card expansion will not work correctly if you use compressed/archived libraries.

Important note #4: TLIB is the most reliable version control system on the market, largely because when TLIB updates a library file, it does not in any way alter or move the existing data. Instead, we just append, in place, to the end of the existing library file. With other version control products, any time you add a new revision to a library file, *all* your old revisions must make a round trip from the disk surface, into RAM (perhaps via a network), get processed by a program, and then be written back to a new location on the disk. In each of those steps, there is a small but finite possibility of data loss. Since TLIB avoids those steps, it also avoids the associated risk of data loss.

However, if you use compressed/archived library files, this reliability advantage is forfeited.

If you *must* use TLIB with archived libraries in a network environment, you can reduce the risk to your data by taking the following precautions:

1) Make sure that all users utilize the same `ARCTEMP` directory, (on a shared file server),

and

2) Instruct all users to watch out for the following PKUNPAK or PKUNZIP prompt:

```
Warning! file XXXX.XXX already exists! overwrite (y/n)?
```

If this prompt is seen, it probably means that another user is accessing the TLIB library or lock file, so TLIB should be aborted with Ctrl-Break.

If you are *certain* that nobody else is accessing the TLIB library or lock file, then the file must have been left in the temporary directory as the result of an aborted TLIB operation. In that case, you can use “`PKZIP F`” to ensure that the `.ZIP` file is up to date, then manually delete the useless temporary file.

Four configuration parameters must be set to enable TLIB to use archived library files. Two of them, **ArcCmd** and **UnarcCmd**, specify the full path and file names of the PKZIP and PKUNZIP programs. The third parameter, **ArcTemp**, specifies a directory for temporary storage of the extracted library and lock files. The fourth parameter, **ArcExt**, is used to specify the file extension of compressed archives (usually `ZIP`).

To use a library which is stored in a `.ZIP` file, simply pretend that the `.ZIP` file is a subdirectory, and specify it in the CP (path) command (or, more conveniently, with the `PATH` configuration parameter).

For example, you could add the following line to your TLIB configuration file to specify that TLIB library files should be stored in the `LIBS.ZIP` archive file, which is in the `C:\TLIB` subdirectory:

```
path c:\tlib\libs.zip\
```

TLIB examines you library file CP (path) setting to determine whether you've specified a directory or an archive file. If "`ArcExt ZIP`" is configured and the path setting ends in "`.zip\`" or "`.zip\name.ext`", then TLIB presumes that the library files are to be stored in a PKZIP archive.

Nested archive files (archives within archives) are not supported. E.g., a library path setting like this would not work correctly:

```
path c:\tlib\xx.zip\yy.zip\
```

Here's an example of how to use the `ArcCmd`, `UnarcCmd`, `ArcTemp` and `ArcExt` configuration parameters. Note that they will not affect the operation of TLIB unless you specify an archive file with the CP command or `PATH` configuration parameter.

```
ArcCmd c:\util\pkzip.exe -es
UnarcCmd c:\util\pkunzip.exe
ArcTemp c:\temp\
ArcExt ZIP
```

If your library files were in an archive file called `LIBS.ZIP`, you would use the CP (path) command to reference the archive just like you would reference a directory, like this:

```
tlib cp c:\libs.zip\ e myfile.c
```

One last note: Because the PKZIP command format may change in future releases, we cannot guarantee permanent compatibility between TLIB and PKZIP.

Customizing the user interface

```
BANNER <1-42>,"string"
NUMBANNER <1-42>
PROMPT <1-42>,"string"
NUMPROMPT <1-42>
HELP <1-49>,"string"
NUMHELP <1-49>
COMMANDS <list-of-commands>
```

The **banner**, **prompt** and **help** parameters allow you to configure the messages which command-line versions of TLIB present to the user. Each of them requires a line number and a string. Unlike the other configuration parameters, `BANNER`, `PROMPT` and `HELP` can be specified repeatedly to specify multiple-line prompt and help messages.

The prompt message is shown to the user when he must select a command. It is usually just one or two lines. The help message is what is displayed when the “`?`” command is selected. It can be up to 49 lines long, but you should limit it to at most 24 lines unless you are sure that users will always utilize display modes that support more than 25 lines.

The initial prompt and help are normally configured by TLIBCONF, but you can easily change them simply by editing your TLIB configuration file.

The number of lines in the banner, prompt and help screen are determined by the `NUMBANNER`, `NUMPROMPT`, and `NUMHELP` parameters, respectively.

Note to users of TLIB 4.12: The `HELP` and `PROMPT` configuration parameters were slightly changed in TLIB 5.0. TLIB 5.0 Removed the extra blank line which TLIB 4.12 displayed before the prompt and help screens. Instead, TLIBCONF now configures the prompt and help screens to have a blank line at the beginning. This change allows you to get rid of the blank line (by changing `TLIB.CFG`), if you wish. We recommend that you let

TLIBCONF (or the TLIB Configuration Wizard) configure your `HELP` and `PROMPT`. Then if you don't like the appearance, you can edit `TLIB.CFG` to change them to your liking.

The **commands** parameter allows you to restrict or expand the set of legal TLIB commands. That is, it lets you configure which TLIB commands the user can use. The user will not be allowed to use any command which is not in the specified list of. Note that TLIB commands are case-insensitive; that is, “N” is equivalent to “n”, etc..

Note: The format of the `COMMANDS` configuration parameter has changed in TLIB 5.

All of the legal commands should be listed in the `COMMANDS` configuration parameter (in TLIB's configuration file, usually `TLIB.CFG`), separated by commas, except that the search mode suffixes may be omitted. For instance, the following enables all of the commands listed above in the “old/new” table on p. :

```
COMMANDS US,EB,C,E,UFK,UD,UK,L,UKS,N,ER,CP,Q,EBS,NS,T,U,\
    UM,CW,ES,?
```

Any commands omitted from the `COMMANDS` configuration parameter will be unavailable to users.

Note that the TLIB Configuration Wizard (or the older TLIBCONF utility) will create a TLIB configuration file with an appropriate `COMMANDS` configuration parameter, prompt, etc., so you probably will not need to manually change it.

New to TLIB 5.01 was the ability to configure the default wild-card search mode for any TLIB command, via an extension to the `COMMANDS` parameter syntax.

To override the default wild-card search mode for a command, simply edit the `COMMANDS` parameter in tlib.cfg, adding “`/x`” to the command, where “`x`” is one of the six legal wild-card search modes.

For example, if your configured `COMMANDS` was:

```
COMMANDS U,UK,US,UD,UKM,UKS,E,ES,EB,...
```

Then you could change the default wild-card search mode for the various U (update) commands to “O” ("owned" files) by adding “/O” to each command, like this:

```
COMMANDS U/O,UK/O,US/O,UD/O,UKM/O,UKS/O,E,ES,EB,...
```

The TLIB Configuration Wizard can utilize this facility to optionally configure TLIB to be more “project oriented.” If you tell the Configuration Wizard to configure TLIB for “project oriented mode” or “ISO 9001 promote levels,” then it will make the default search mode for most commands be “T” or “A” (to search the project level), and you'll need to first add your source files to the project level via the “A” command before you can extract and/or update them (unless you override the search mode). (See also the FIND1FILE parameter, p. 306.)

The COMMANDS configuration parameter can be continued onto additional lines in the configuration file, by adding a trailing backslash (“\”) character. (COMMANDS is the *only* configuration parameter which can be continued in this way.)

Input lines in TLIB's configuration file should never exceed 254 characters in length. However, it is possible for a COMMANDS parameter to be too long to fit on a 254-character line, so we've added a way to extend it to the desired length.

For example, these two COMMANDS parameters are equivalent:

```
COMMANDS U/O,UK/O,US/O,UD/O,UKM/O,UKS/O,E,ES,EB,EBS,L,T,H,Q

COMMANDS U/O,UK/O,US/O,UD/O,UKM/O,UKS/O\
         ,E,ES,EB,EBS,\
         L,T,H,Q
```

Note that leading whitespace (tabs and/or blanks) is ignored on the continuation lines.

Important: as listed in the COMMANDS configuration parameter, the TLIB commands must have their suffix characters in alphabetical order. Thus, this is okay:

```
COMMANDS U/O,UK/O,US/O,UD/O,UKM/O,UKS/O,E,ES,EB,EBS,L,T,H,Q
```

but this is in error:

```
 COMMANDS U/O,UK/O,US/O,UD/O,UMK/O,UKS/O,E,ES,EB,ESB,L,T,H,Q
```

because `UMK` should be `UKM`, and `ESB` should be `EBS`.

Be careful: the current edition of TLIB doesn't do much validity checking of the `COMMANDS` parameter, so it will not detect this error.

For your convenience, you may equate single-character shorthand synonyms for any of the multi-character commands. One use for this is to let you make TLIB 5 look and feel like earlier versions of TLIB (except that the user will still have to press `ENTER` after each command).

TLIBCONF will offer to configure the `COMMANDS` parameter to mimic most of the single-character commands in earlier versions of TLIB. This is helpful for users who have become accustomed to the old-style commands, or who have editor macros or batch files that use the old single-letter commands, but there seems to be universal agreement among our customers that the new style is much better.

Note that you may *not* redefine the multi-character commands.

Also, to avoid confusion, we recommended that you not redefine any of the twelve default single-character commands to mean something strange. The default single-character commands are: A,C,E,F,L,M,N,Q,S,T,U,?.

Note that all but four of these commands have the same meanings that they had in TLIB 4.12. The exceptions are A, F, S and M.

“A” was “update with specified version number” in TLIB 4.12, but is “add or alter project level” in TLIB 5. The “update with specified version number” command is now “US”.

“F” was “fast-update/freshen” in 4.12, but is “filter file names” in TLIB 5. The “fast-update/freshen” command is now “UF” or “UFK”.

“S” was “split-library” (create new library with specified starting version number) in 4.12, but is “snapshot-version-label” in TLIB 5. The “split-library” command is now “NS”.

“M” was “make branch” (update, specifying version number, but do not unlock) in TLIB 4.12. The equivalent TLIB 5 command is “UKS”. The “M” command is “migrate changes” in TLIB 5.xx.

The following configuration parameter gives single-character shorthand equivalents for many of the possible commands, chosen so that this new version of TLIB will respond correctly to most of the old TLIB 4.12 commands.

```
COMMANDS US,B=EB,C,E,UFK,I=UD,K=UK,L,UKS,N,O=ER,P=CP,Q,\
         R=EBS,T,U,W=CW,X=ES,F,?
```

For a fancier display (with highlighting, multiple colors, or whatever), you can embed ANSI escape sequences in your prompt and help lines and use DOS's `ANSI.SYS` device driver (see your DOS manual for details on `ANSI.SYS` and the ANSI escape sequences). We've included a simple AWK program named `ANSIFY.AWK`, which can add ANSI highlighting to the prompt and help which were configured in `TLIB.CFG` by TLIBCONF. If you don't already have an AWK, you can use Rob Duff's freeware AWK (from the `PUBLIC.ZIP` archive on the TLIB distribution diskette).

To add ANSI highlighting, use the following command:

```
awk -fansify.awk tlib.cfg >tlib2.cfg
```

Then test the new configuration file like this:

```
tlib c "include tlib2.cfg" ?
```

If you are happy with the result, you can replace `TLIB.CFG` with the newly created `TLIB2.CFG`.

`BANNER` *<1-42>*`,"`*string*`"`
`NUMBANNER` *<1-42>*

These parameters let you create a customized TLIB “startup banner,” which you can configure to display whatever you wish. It works just like the `PROMPT` and `HELP` configuration mechanisms, except that (unlike the prompt & help screens) the `BANNER` is always displayed (unless suppressed with the `-q2` command line option, p.), and (unlike the prompt) it is displayed only once (when TLIB starts up).

The intended purpose is to provide a convenient mechanism for a System Librarian to provide helpful advice and instruction to the rest of the TLIB users.

There are two configuration parameters, `NUMBANNER` and `BANNER`, to configure your customized startup banner. `NUMBANNER` specifies the number of lines in your customized banner. `BANNER` specifies the individual lines. Example:

```
numbanner 6
banner 1,”
banner 2,'Hi, %TLIBCFG:id%.  Current projlev is %TLIBCFG:projlev%.'
banner 3,"Don't forget to mention the problem numbers in your TLIB"
banner 4,'comments when you store a new version of a source file!'
banner 5,'Call me at extension 234 if you have questions.  -Dave'
banner 6,”
```

Note the use of double quotes on the third line, to allow quoting the apostrophe.

Huge File Support

`MULTIPASS` *<Y/N>*
`MAXLINES` *<100-16380>*
`PASSSIZE` *<100-16380>*

When running with the default configuration parameters, TLIB processes source files of up to 16000 lines (or binary records) in length in a single pass. Larger files are handled with multiple passes, which permits TLIB to manage files of nearly unlimited size.

(*Note:* The real-mode DOS version of TLIB has a default `PASSSIZE` of 4000, instead of 16000.)

TLIB's memory footprint can be reduced, at the expense of both performance and storage efficiency, by using a smaller `PASSSIZE` setting. But this is not recommended for most users.

The only significant problem with using the default `PASSSIZE` (16000) is that the increased memory requirements may prevent the real-mode DOS version of TLIB from being able to process large files. This is probably not a significant problem for you; you can solve it simply by running one of the protected-mode versions of TLIB (e.g., `TLIBX.EXE` or `TLIB2.EXE` or `TLIB32C.EXE` or *TLIB for Windows*) instead of the real-mode DOS version.

Mostly for historical reasons, it is also possible to disable TLIB's multipass operation, by configuring **Multipass N**, in which case TLIB will be unable to process files that are too long to be handled in one pass. This is not generally recommended.

Note: For `FILETYPE BINARY`, and `FILETYPE RUNLEN`, TLIB uses an internal “record” representation which is analagous to the lines in a text file, except that TLIB uses an adaptive algorithm to delimit the “records,” instead of being hard-coded to look for carriage-return/line-feed.

If you Run TLIB under OS/2, you should use `TLIB2.EXE` for command-line work. Under DOS or Windows 3.1x, use the DOS-extended version of TLIB, `TLIBX.EXE`, for command-line work, rather than the real-mode `TLIBDOS.EXE`. Under modern versions of Windows, use the 32-bit TLIBs: `TLIB32C.EXE` and the 32-bit Windows GUI version.

For users of the DOS versions of TLIB, *the preferred method* of handling large files is to configure `PASSSIZE 16000` and `MULTIPASS Y` and run the DOS-extended TLIB, `TLIBX.EXE`.

If `MULTIPASS Y` is configured, TLIB is able to handle arbitrarily large source source files, by processing them with multiple “passes” of `PASSSIZE` lines (or binary records) each. The additional passes will degrade performance somewhat, but only for files of more than `PASSSIZE` lines; shorter files will be processed just as efficiently as before.

If you have plenty of extended memory available (and a DPMI, VCPI, or XMS memory manager, such as that provided by Windows, EMS386, or HIMEM), and you need to handle many extremely large files, then the simplest and most efficient solution is to configure `PASSSIZE 16000` and `MULTIPASS Y`, and run `TLIBX.EXE` instead of the regular DOS `TLIB.EXE`. However, when processing small files, `TLIBX.EXE` is somewhat slower than the regular, real-mode `TLIB.EXE` (and much slower than `TLIB2.EXE` under OS/2 or `TLIB32C.EXE` under modern versions of Windows).

Note #1: to decrease `TLIB`'s memory usage, you must *decrease* the `PASSSIZE` (or `MAXLINES`) setting. This slows down TLIB somewhat, by increasing the number of passes required, but it cuts memory requirements by reducing the number of lines that are kept in memory at one time.

Note #2: These paramters affect the fundamental structure of the TLIB libraries, so if you change them to make TLIB work with a large source file, you must delete the TLIB library and create a new one with the N command before the new settings take effect (or convert it with `TLIBTLIB.EXE` or `TLIBTLIB.AWK`, see p. 296). Simply changing `PASSSIZE` (`MAXLINES`) will not work!

TLIB divvies up the available RAM memory into two kinds of buffer area: the line buffer, and the text buffer. The line buffer requires 8 bytes per source file line, or 32,000 bytes for a 4000 line buffer. This storage is used even if your file is less than 4000 lines long. The text buffer uses the rest of the available RAM; the “fullness” of the text and line buffers is indicated by the “memory % used...” messages which the real-mode version of TLIB often displays.

The `PASSSIZE` (or `MAXLINES`) parameter provides you with a way to change the size of the line buffer. However, increasing the line buffer size will reduce the amount of memory left for the text buffer. Conversely, decreasing the `PASSSIZE` parameter will free up some additional memory for the text buffer. If you are using the real-mode version of TLIB, you may need to decrease the `PASSSIZE` parameter if your computer does not have very much RAM memory. The `PASSSIZE` parameter can be set as high as 16380, but this is only useful if your source files are made up of extremely short lines, unless you are runnning one of the the protected mode editions of TLIB. 7000-8000 is the practical limit for typical program source code on a 640K PC, and 3000-4000 is good choice for most users of the real-mode DOS version of TLIB.

Note #3: Large source files with unusually long lines will generally require that you reduce `PASSSIZE` to handle them with the real-mode DOS version of TLIB. The strategy is to choose a `PASSSIZE` setting which will result in a “reasonable” portion of the source file being handled at one time, e.g. 100-150kb. A “typical” source file has an average line length of around 20-35 bytes. With the default `PASSSIZE` setting of 4000 lines, this implies that up to 140kb will be handled per pass (35 `x` 4000 = 140,000). However, if you have large source files with lines that average considerably longer than that, you'll need to reduce PASSSIZE (MAXLINES) to manage them with the real-mode DOS version of TLIB. (An easier/better solution is usually to use a protected-mode version of TLIB such as TLIBX.EXE.)

Example: suppose you have a 500kb source file, which is 7500 lines long. The average line length is 75 bytes (500,000 / 7500 = 77). To make TLIB handle the file in four “chunks” of about 125kb each, you would need to set PASSSIZE to a little over 1/4 of 7500, i.e. about 2000,. With PASS-SIZE 4000, TLIB would try to handle the file in only two passes (of 4000 and 3500 lines, respectively), which would require that TLIB be able to process over 250kb at a time. (This is not a problem for protected-mode versions of TLIB, but it *might not* work with TLIBDOS.EXE unless you have an unusually large amount of free “low” memory. Note that TLIB's memory requirements increase somewhat as the library file grows.)

Note: For FILETYPE BINARY, the adaptive algorithm TLIB uses generally results in an effective internal record size (or “granularity”) of about 20-30 bytes. Thus, for FILETYPE BINARY it is rarely necessary to change the PASSSIZE (or MAXLINES) setting to handle small binary files. But to most efficiently handle binary files of more than about 80K, you should use a protected-mode version of TLIB, such as TLIBX.EXE, and configure PASSSIZE 16000.

Users of the real-mode DOS version of TLIB can obtain a good idea of the best choice for PASSSIZE by creating and updating an experimental library file for one of your largest source files (*important:* do this test with MUL-TIPASS mode disabled, i.e., “MULTIPASS N”). TLIB will display the percentage of the text and line buffers which was used, like this:

```
 Memory 38% used up.  Line buffer 59% full.
```

We recommend that you choose a PASSSIZE value which results in the line buffer being more nearly used up than the text (“memory”) buffer; the 38%-59% example, above, is fine. If you find that the “memory % used” approaches or exceeds the “line buffer % used” with a default (4000 line) buffer, then you should reduce the PASSSIZE parameter. You'll still be able to handle large files if you enable MULTIPASS Y.

If TLIB runs out of text buffer before filling the line buffer, it will resort to a “garbage collection” process to try to handle the library file. If that fails, TLIB will display an “Out of memory” error message and abort the operation. So be very sure to select the PASSSIZE (or MAXLINES) parameter conservatively!

Important: once you have created a library file, you cannot later reduce TLIB's memory requirements for that library file by reducing the `PASSSIZE` (or `MAXLINES`) parameter, since the `PASSSIZE` value is stored in the library file, and affects its structure. You can, however, use the DOS-extended version of TLIB, `TLIBX.EXE`, instead of `TLIB.EXE`. Or, you can convert the library file to have a smaller `PASSSIZE`, with the `TLIBTLIB.EXE` (or `TLIBTLIB.AWK`) conversion utility (see p. 296). Or, in a pinch, you could split the library file with the `NS` command (see p. 239).

Example: Here we set `MULTIPASS` to enabled (Y), and set `PASSSIZE` (`MAXLINES`) to 4000 lines, which are the defaults for the real-mode DOS version of TLIB, `TLIBDOS.EXE`:

```
multipass Y
passsize 4000
```

Cmpr and Tlmerge/Diff3 parameters

CMPR is the TLIB stand-alone "delta" generator (compare utility). The output is a "differences" file.

DIFF3 is a 3-way compare utility which will combine two independent sets of changes to the same source file. The output is a new source file with both sets of changes. Note that DIFF3 is *not* compatible with the Unix™ utility by the same name.

For how to use these tools, see pages 225 and 233.

There are several configuration parameters for adjusting the behavior of these two tools:

`ADDCTRLZ` *<Y/N>*

The **addCtrlZ** configuration parameter is used by CMPR and DIFF3, as well as by TLIB (see p. 276).

`CMPRENTAB` *<Y/N>*
`CMPRDETAB` *<Y/N>*
`DIFFLINES` *<100-16380>*

These three configuration parameters are used only by CMPR and DIFF3. The **CmprEntab** and **CmprDetab** parameters affect the treatment of tabs and blanks. The **DiffLines** parameter determines the internal line buffer size for CMPR and DIFF3 (like `MAXLINES` does for TLIB).

CmprEntab determines whether multiple blanks will be converted to tabs before the comparison is performed. **CmprDetab** determines whether tabs will be expanded to blanks in the output file.

Both of these default to `N` (disabled). If you set them to `Y` (enabled), tabs are equivalent to blanks, and there will be no tab characters in the output. If you set `CmprEntab Y`, but leave `CmprDetab N`, then the output will be smaller because blanks will be converted to tabs wherever possible.

If you leave both `CmprEntab N` and `CmprDetab N`, then CMPR will not perform any blank/tab conversions, and lines containing tabs will not be considered the same as lines which appear identical but contain only blanks.

The fourth combination of parameters, `CmprEntab N` and `CmprDetab Y`, is legal but not very useful.

If each of the input files contain at most **DiffLines** lines, then CMPR and DIFF3 will handle them in one “pass.” Otherwise, the files will be handled in several pieces (which is slightly slower and occasionally less “robust”).

Multi-pass operation works well with both DIFF3 and CMPR, unless there are really massive differences between the files. The default is `DiffLines 3000`.

Here we set all three parameters to the defaults:

```
 CmprEntab N
 CmprDetab N
 difflines 3000
```

`D3COLLIDE` *<line-to-insert>*
`D3FLAG2` *<0-253>*`,"`*<string>*`"`
`D3FLAG3` *<0-253>*`,"`*<string>*`"`

DIFF3 uses the same configuration parameters as CMPR, plus three of its own: **D3collide**, **D3flag2** and **D3flag3**.

D3collide changes the special line to be inserted in the output file wherever DIFF3 detects conflicting changes in file2 and file3. It should be something which is easy to search for with a text editor. The default will be fine for most users (scan for the "`###`"):

```
 D3collide /* ###Change collision detected! %n */
```

The "`%n`" in the flag line is replaced by the file name(s) for the changed files when the flag is inserted in the output file.

Because changes to a program may well conflict even if no single line in the base file is altered in both file2 and file3, you may wish to flag all changes, rather than just those which happen to "collide." **D3flag2** and **D3flag3** are provided for this purpose. To flag all lines which were modified or inserted in file2 and in file3, respectively, you could define:

```
 D3flag2 68,"/*###file2*/"
 D3flag3 68,"/*###file3*/"
```

The string "`/*###file`*x*`*/`" will be appended to each changed or added line, so that you can easily find the changes by searching with a text editor. The "68" is a column number, which guarantees that the string will start at column 68 or higher. That is, if the line is less than 67 characters long, it will be padded with blanks before the string is appended.

If you set the `D3FLAG2` & `D3FLAG3` columns to 1, then the string will just be appended to the end of the line, with no blank padding. If you prefer to have the special string inserted at the beginning of the line, rather than the end, then you can specify column 0 (zero).

PCOM & PCOMS parallel port file transfer

PCOM is a simple MS-DOS utility for very high speed PC-to-PC file transfer via back-to-back connected parallel printer ports, using a special cable (not a “laplink cable”). PCOM is useful for transferring data between computers which are equipped with incompatible disk drives (i.e., 5.25" and 3.5").

PCOM transfers data with an effective baud rate of 79-2000 Kbaud, depending upon the speed of the computers which use it. The speed is roughly proportional to the CPU speed of the slower of the two compters (about 7900 bytes/sec on a 4.77 MHz PC).

Because the data transfer is fully handshaked, it is highly reliable and is completely immune to “missed interrupt” (overrun) errors. Because PCOM does not disable interrupts, it generally will not interfere with interrupt-driven background tasks, such as communications programs, print spoolers, local area networks, etc..

PCOM runs on any IBM-PC or compatible MS-DOS computer which has a parallel printer port.

Abbreviated Operating Instructions

for more extensive instructions (on-line help):

```
PCOM
```

to test the printer ports & reconfigure PCOM:

```
PCOM TEST
```

to run as a “server” for the other computer:

```
PCOM [port] SERVER
```

to see a directory list of files on the “server” computer:

```
PCOM [port] X:<wild-card-spec>
```

to send file(s) to the “server” computer:

```
PCOM [port] <fromfile> X:<tofile> [options]
```

to receive file(s) from the “server” computer:

```
PCOM [port] X:<fromfile> <tofile> [options]
```

All names can contain wild-cards and/or drive/directory specifications. Multiple and leading asterisks are handled properly, but you may not use file lists. For example:

```
PCOM X:a:\*ame.* *.*
```

To exit PCOM at any time, press Ctrl-Break.

Options

The `[port]` option is needed only if you wish to override the default choice of printer ports. You can specify it either as a printer device (`-pLPT1`, `-pLPT2` or `-pLPT3`), or as a hexidecimal address (`-p378`, `-p278` or `-p3BC`).

The other options are:

`-C` Calculate and check **C**RCs

`-M` **M**ake subdirectories if necessary (often used in combination with `-S`)

`-S` Recurse through tree of **S**ubdirectories

`-N` Transfer only **N**ewer files

`-T` Just **T**est, don't really transfer files

If you specify `-C`, then after each file is transferred, PCOM will close the file, reopen it, read it from the disk, and calculate a 16-bit CRC checksum. If the checksum is incorrect, PCOM will display an error message. This option offers an extra measure of safety to ensure that the file was correctly transferred and that it can be read from the disk without error. Of course, the file transfer will be a bit slower.

Normally, PCOM will replace already existing files without warning, just like the DOS copy command. However, if you specify the `-N` option, PCOM will only replace a file which is "older" than its replacement. For most files, the determination of which file is "older" is done in the obvious way, by checking the file dates.

Special -N handling of TLIB libraries

PCOM has special handling which is used when you copy TLIB libraries with PCOM's `-N` option. (PCOM detects whether a file is a TLIB library by checking whether the first two bytes are "`.V`".) For TLIB libraries, the "newness" depends, not upon the file dates, but upon the file sizes. Usually, the shorter of two identically named TLIB library files is "older." (Since TLIB always appends new deltas to the end of a library file, the library file can only grow.) But if both library files have been updated, this is not true; neither of them can be considered "newer," since replacing either file with the other would cause a loss of data. So, in addition to comparing the file sizes, PCOM compares a CRC checksum for the smaller library file to a CRC calculated from the first part of the larger file, to determine whether the shorter library file contains a subset of the versions in the larger one. If not, then the shorter file may contain versions not present in the longer ("newer") file. If PCOM detects this situation, a warning is issued and the file is not copied; the warning alerts you to the fact that you need to compare and reconcile the new versions in the two libraries (DIFF3 can help with this chore).

This feature is helpful if you take work home on a lap-top computer. You can copy the library files to your lap-top computer, take it home and do your work, and then, when you return, you can safely update the "at-work" versions of the library files even if you hadn't checked the modules out for modification. (Warning: PCOM does not understand TLIB's check-in/out locking mechanism, so it does *not* warn you if someone else has a file checked-out for modification.)

PCOMS - a background PCOM server

The trouble with running PCOM SERVER as described above is that it monopolizes the “server” computer. To avoid this problem, we have developed a memory resident (TSR) version of the PCOM server, `PCOMS.EXE`. It runs in the background, using up about 31K of RAM memory, and allows you to continue to use your “server” computer for other things, even as you access its disks via PCOM.

To run PCOMS as a background server for the other computer:

```
 PCOMS [port]
```

If you have purchased multiple copies of TLIB, or a multi-station license, you can even run two or three copies of PCOMS on the “server” machine, to simultaneously service two or three other computers connected to two or three printer ports. (To configure PCOMS for such an environment, you could run `PCOM TEST` to configure PCOMS for one port, then save `PCOMS.EXE` under another file name, then switch cables, run `PCOM TEST` again, etc..)

PCOM 5.00k changes

PCOM now supports additional options; run it without parameters for help.

PCOM and PCOMS now have improved support for non-standard printer ports, and an improved `REMOTE` (“remote control”) mode.

Also, PCOM 5.00k now supports the following extended wild-card syntax:

```
        PCOM x:wild-card-spec d:*\*.* or
        PCOM wild-card-spec x:d:*\*.*
```

where “*d:*” is a drive specification. This syntax tells PCOM to copy files to the same directory and file name, except onto a different drive letter. If the “*d:*” is left off, then the destination file will go to the same drive letter as the source file:

```
        PCOM x:wild-card-spec *\*.* or
        PCOM wild-card-spec x:*\*.*
```

Note that PCOM is a DOS-only program (though it also runs in the DOS boxes of OS/2 1.3 and later).

There are a few idiosyncrasies when running PCOM under OS/2 2.x:

a) If you use OS/2 2.x, you'll have to use the “`-p`” option to tell PCOM which printer port to use; the default is generally wrong. Alternately, you can run “`PCOM TEST`” on both computers (simultaneously) to re-configure PCOM.

b) Under OS/2 2.x, PCOMS (the TSR server) is slow. It may be that one of OS/2's VDM (DOS box) settings could be adjusted to solve this. Or, just run “`PCOM SERVER`”.

c) Under OS/2 2.x, PCOM may often pause before exiting. This is due to a problem in the OS/2 printer port driver, which was fixed in later versions of OS/2.

d) “`PCOM SHELL`” doesn't work under DOS 5 or 6 or an OS/2 “DOS box.” However, remarkably enough, it seems to work fine under native MS-DOS 4.01 running in an OS/2 2.x VDM.

Examples

`PCOM -p3BC *.txt x:c:\*.* -c` (copy using port 3BCH, with CRCs)

`PCOM c:\*.* -s` (list *all* files on remote computer's C: drive)

`PCOM -pLPT1 *.txt x:c:\z\*.* -m` (copy via `LPT1`, & “`mdir c:\z`”)

`PCOM x:\*.* p\*.* -m` (copy from server using default, with “`mdir p`”)

`PCOM c:\*.* x:c:\*.* -s -m` (copy whole disk, except hidden files)

`PCOMS -pLPT1` (install background server on LPT1)

`PCOM -pLPT1 REMOTE` (remote-control the server's screen & keyboard)

Configuration

Connect the special printer cable between the parallel printer adapters of the two computers.

PCOM is initially set up to use your last printer port (probably address 378 hex if you have two of them). To use a different printer port, you can use the `-p` command line option to select the port, or connect the cable and run “`PCOM TEST`” on both computers.

If you run `PCOM TEST`, PCOM will offer to patch itself (and PCOMS), if necessary, to use the correct default port. If you are using DOS 2.x, `PCOM.EXE` and `PCOMS.EXE` must be in the current directory when you do this.

You can also patch the printer port address manually. The value to be patched is near the end of the `PCOM.EXE` and `PCOMS.EXE` files, and is immediately preceded in the file by the string “`port=`”. Normal values are 378, 3BC or 278 hex.

Note that you can always override the default by specifying the “`-p`” (port) option on the PCOM command line.

Copyright/license

PCOM is not of much use when run on only one machine at a time! So we're making an exception to the copyright/license terms for the single-user version of TLIB, to allow you to use PCOM on *two* computers at a time. This applies only to PCOM, not to the other programs in the TLIB package.

This product uses the TesSeRact™ Ram-Resident Library and supports the TesSeRact Standard for Ram-Resident Program Communication. TesSeRact is a trademark of the TesSeRact Development Team. TesSeRact information could formerly be obtained from The TesSeRact Development Team, c/o Innovative Data Concepts (a Pennsylvania company), Chip Rabinowitz, President. Innovative Data Concepts sold an improved, commercial version of TesSeRact, called the “TesSeRact RAM Resident

Development System.” Unfortunately, the TesSeRact Development Team has disbanded, IDC has gone out of business, and we are unable to locate Chip.

Cable

PCOM needs a special cable to connect the printer ports of the two computers. The cable requires eleven wires and two 25-pin male “D” shell connectors. It should be wired like this:

I/O	name	pin	pin
out	DB3	5	15
out	DB4	6	13
out	DB5	7	12
out	DB6	8	10
out	DB7	9	11
in	-ACK	10	8
in	-BUSY	11	9
in	+PE	12	7
in	+SLCT	13	6
in	-ERR	15	5
	GND	18-25	18-25

Actually, you usually do not need to attach GND to all eight ground pins... just pin 18 is enough for most computers. For minimum RFI radiation, you should also attach the cable shield to GND at both ends.

We've seen PCOM work with cables up to 75 feet long, but we recommend that you try to keep the length below 25 feet, especially in “high noise” environments. If the cable is more than 10 feet long, we recommend that you use “low capacitance” wire.

The cable is completely symmetrical. It does not matter which end is connected to which computer.

For those who prefer not to make their own cables, we stock six foot PCOM cables for $30.00 (plus shipping, 1 lb.).

Expandir

EXPANDIR is used to pre-allocate disk subdirectory space, for improved performance.

The problem is that when a DOS subdirectory grows beyond a single disk cluster in length, a second cluster will be allocated, normally far from the first. Thus, multi-cluster subdirectories are generally severely fragmented. This is why file access tends to be so slow in subdirectories which contain many files: each time an operation is done which requires accessing the directory (e.g., an “open”), a lengthy “seek” operation must be done for each directory cluster.

EXPANDIR forces DOS to add the needed directory clusters ahead of time, which usually results in a contiguous or near-contiguous directory, for much improved disk performance. A hard disk with 2K clusters can hold 62 files in the first cluster of a subdirectory, and 64 files in each additional cluster. Thus, you may wish to use EXPANDIR on any subdirectory in which you plan to store 63 or more files.

EXPANDIR is intended for use only on “normal” DOS subdirectories. It will not work on “root” directories (which are of fixed length), and it is unlikely to improve performance on WORM optical drives or Novell network file servers.

For instructions on how to use EXPANDIR, run it with no parameters.

Testlock

TESTLOCK is our tool for testing network and OS network file-sharing, locking and related functions.

TESTLOCK is mostly used in either of two automatic test modes:

`AUTO`	tests file sharing/locking on a single machine
`AUTO2`	uses two machines for a more thorough test

We include five versions of the TESTLOCK executable:

`TESTLOKR.EXE`	-	real-mode DOS
`TESTLOKP.EXE`	-	16-bit OS/2+NT+DOS ("bound" family-mode)
`TESTLOK2.EXE`	-	32-bit OS/2
`TESTLK95.EXE`	-	real-mode Win-95 (uses Int 21H LFN functions)
`TESTLK32.EXE`	-	Win-32 (for Windows-9x/Me/NT/2K/XP)

All five versions are functionally identical, except that they were compiled and linked for different operating environments.

Most versions of TESTLOCK will run under more than one PC operating system. For the most thorough test of your operating system and network file I/O, we recommend that you test it with all versions of TESTLOCK which will run on your operating system. Some network client drivers have bugs which only affect certain subsystems. Thus, it is not safe to assume that your network software is working correctly after testing with just one version of TESTLOCK.

This table shows which versions of TESTLOCK can be run under each of several common PC operating systems:

	TESTLOKR	TESTLOKP	TESTLOK2	TESTLK95	TESTLK32
MS-DOS	Yes/8.3	Yes/8.3	No	No	No
Windows 3.1	Yes/8.3	Yes/8.3	No	No	No
Windows 95	8.3	8.3	No	Yes	Yes
Windows NT	8.3	Yes	No	No	Yes
OS/2 1.3	8.3	Yes	No	No	No
OS/2 2.x	8.3	Yes	Yes	No	No
OS/2 Warp	8.3	Yes	Yes	No	No

Where:

`Yes`	means that it runs
`8.3`	means that it runs, but only using short file names
`No`	means that it won't run

The various versions of TESTLOCK test the I/O methods used by various versions of the TLIB Version Control System:

Program	Does I/O similarly to...
TESTLK32	Win-32 versions of TLIB
TESTLK95	TLIBX.EXE and 16-bit TLIB for Windows
TESTLOK2	(most 32-bit OS/2 programs – not any version of TLIB, though)
TESTLOKP	TLIB2.EXE
TESTLOKR	TLIBDOS.EXE

Common Usage:

```
     {PROGRAM} {PATH} {OPTION}
```

where:

{PROGRAM} is one of the 5 versions of TESTLOCK

{PATH} is *not* optional, and should generally be in one of these forms:

```
     d:\subdir\
     \\server\vol\subdir\
```

{OPTION} is *not* optional, and must one of the following:
AUTO tests file sharing/locking on a single machine
AUTO2 uses two machines for a more thorough test

Example:

```
      testlokr c:\ auto2
```

For more detailed instructions, see the file `TESTLOCK.TXT`.

Touch

Purpose: Set a file's date & time to “now”

Usage: `TOUCH filename`

where filename is the path of the file whose date/time you wish updated. You can use wild-cards and file lists to specify multiple files.

TOUCH is a little program to change the “last modified” date and time for a file to be the current date and time. It is sometimes handy when you are using MAKE, since it provides a convenient way to force a subsequent build (compile, link or whatever). Just “touch” any file in the dependency list(s) (to the right of the colon) for the build step(s) which you want to force.

TOUCH has a couple of advantages over simply editing a file and then immediately saving it without any changes (which will also set the date and time to “now”). For one thing, TOUCH can be used on any file, not just text files.

Also, TOUCH will not set the DOS “archive” attribute, which would cause the touched file be needlessly saved again in your next incremental backup (most backup utilities, including DOS backup, use the archive attribute to decide which files need to be saved). You *do* make regular backups, right?

And, of course, TOUCH is very fast, since the file is not actually copied or modified in any way.

Make

Another difficulty faced by developers using multiple source code files is not really related to library maintenance. It is the problem of doing the compiles, links, etc. needed to keep the `.OBJ`, `.EXE`, etc. files up to date with the source files.

Suppose you have an “include file” which is used by many of your source files, and you make a change to it. Which source files should you re-compile, and which `.EXE` modules should you re-link?

The developers of UNIX™ hit upon an elegant solution: create a file (called a “makefile”) describing the dependencies (e.g., whicch `.OBJ` files depend upon which source files). Then a program can examine the dependencies and the date/time each file was last modified to determine which compile and link commands are needed.

The program was called MAKE, and today there are many companies selling MAKE utilities, most of them bundled with other programs.. Burton Systems Software does not sell a MAKE utility. However, we cooperate with another vendors who sell an excellent MAKE utility, and we also give our customers a copy of a primitive public domain MAKE program written (mostly) by Mr. Landon Dyer.

Dyer MAKE is not copyrighted, so you may do anything you wish with it. However, he does ask that you please do *not* sell it, either with or without modification. Please respect his wishes.

Dyer MAKE is supplied with full source code (in *Microsoft*™ *C 6.0*, ported from Borland's *Turbo-C*™ and (before that) *Lattice*™ *C 2.13*). Incidentally, Landon tells us that it can also be compiled and run under VAX/VMS.

Note #1: MAKE can be used to update your TLIB library files whenever you change your source files. However, it is simpler and faster to use TLIB's UF (fast update) command, which has the same effect.

Note #2: Landon Dyer MAKE is a very “vanilla” tool. It lacks some of the features of Unix MAKE (we especially miss “inference rules”), and it lacks the marvelous “automatic dependency generators” provided by the better commercial MAKE utilities, such as *Opus MAKE*. We cooperate with Opus Software to ensure that Opus Make and TLIB Version Control work together well.

Opus Software (*Opus*™ *MAKE* and *MKMF*)
http://www.opussoftware.com/
1032 Irving St., Suite 439-B, San Francisco, CA 94122
Phone: (415) 485-9703   Fax: (415) 485-9704

Syntax

The format of the MAKE command for Dyer MAKE is:

```
MAKE [-N] [-A] [-F file] [name ...]
```

where:

`-F:`	*use* file *instead of default* makefile
`-A:`	*assume all modules are obsolete (rebuild everything)*
`-N:`	*don't recompile, just list steps to recompile*
`name:`	*module name(s) to recompile*

Files

Dyer MAKE consists of the following files: `MAKE.BAT` (or `MAKE.CMD` under OS/2), MAKEEXE.EXE, and `DELBAT.BAT` (plus the source code). When you use MAKE, these three files should be in either the current directory or in a directory mentioned in your PATH.

A temporary file called `MAKE$$$$.BAT` is created by MAKE; this file will be deleted when MAKE has finished (unless `DELBAT.BAT` is missing).

Description

MAKE is a utility inspired by the Unix™ command of the same name. MAKE helps maintain programs that are constructed from many files.

MAKE processes a “makefile”, which describes how to build a program from its source files, then produces and runs a `.BAT` (“script”) file containing the commands necessary to re-compile the program. (Note: “script” is Unix lingo for “`.BAT`”)

Be careful: this MAKE is *not* compatible with Unix™ MAKE!

The “`-N`” option causes MAKE to print out the steps it would follow in order to rebuild the program. The “`-A`” option tells MAKE to assume that all files are obsolete, and that everything should be re-compiled. The “`-F`” option, followed by a filename, can be used to specify a makefile other than the default one.

If no names are specified on the command line, the first dependency in the makefile is examined. Otherwise, the specified root names are brought up to date (the root names are the things you want rebuilt; that is, the names you specify on the DOS command line).

The default makefiles are:

```
MAKEFILE
..\MAKEFILE
```

If the first makefile cannot be found, MAKE attempts to use the next one. If no makefile is ever found, MAKE prints a diagnostic and aborts.

The Makefile

Comments begin with “`#`” and extend to the end of the line. A “`#`” (or almost any other character) may be escaped with the escape character, the backquote (`` ` ``). An escape character may be typed by doubling it (``` `` ```). The standard C language escape codes are recognized:

`` `n `` ASCII 10, line feed
`` `r `` ASCII 13, carriage return
`` `t `` ASCII 9, tab
`` `b `` ASCII 8, backspace
`` `f `` ASCII 12, form feed

A makefile is a list of dependencies. A dependency consists of a root name, a colon, and zero or more names of dependent files. (The colon MUST be preceded by whitespace.) For instance, in:

```
make.exe : make.obj parsedir.obj file.obj mk.h
```

the file “`make.exe`” depends on four other files. A root name with an empty dependency, as in:

```
print :
```

is assumed *never* up to date, and will always be re-compiled.

The dependency list may be continued on successive lines:

```
bigfile.exe : one.obj two.obj three.obj
four.obj five.obj six.obj gronk.obj
freeple.obj scuzzy.lnk frog.txt greeble.out
```

Any number of “method” (DOS command) lines may follow a dependency. Method lines begin with whitespace (blank or tab). When a file is to be re-compiled, MAKE copies these method lines (minus the leading blanks or tab) to the script (`.bat`) file. For example, in:

```
make.exe : make.obj parsedir.obj file.obj macro.obj mk.h
         link make,parsedir,file,macro
         echo "Another version of MAKE..."
```

the two lines following the dependency make up the method for re-linking the file “`make.exe`”.

If the macro “`~INIT`” is defined, its text will appear first in the script file. If the macro “`~DEINIT`” is defined, its text will appear last in the script file. By defining these two macros, it is possible to set the directory or whatever:

```
~INIT = echo on`ncd \workdir`nif not exist
qwarkle.xyz goto exit   (should be all 1 line)
~DEINIT = :exit`ncd \
~DEINIT = $(~DEINIT)`necho "Done."
```

this will expand (in the script file) to:

```
echo on
cd \workdir
if not exist qwarkle.xyz goto exit
 . . .
```

```
:exit
cd \
echo "Done."
```

When a root's method is defined, the value of the macro "`~BEFORE`" is prefixed to the method, and the value of the macro "`~AFTER`" is appended to it.

Frequently one wants to maintain more than one program with a single makefile. In this case, a "master dependency" can appear first in the file:

```
allOfMyToolsAndHorribleHacks : cat peek poke.exe grunge
cat : cat.exe
cat.exe : (stuff for CAT.EXE)
peek : peek.exe
peek.exe : (stuff for PEEK.EXE)
poke.exe : (stuff for POKE.EXE)
grunge : grunge.com
grunge.com : (stuff for grunge)
```

In other words, MAKE will bring everything up to date that is somehow connected to the first dependency (the incredibly lengthy filename specified in this example can't actually exist).

Macros

A macro is defined by a line of the form:

```
<macro-name> = <macro-body>
```

The \`=' MUST be surrounded by whitespace. A macro may be deleted by assigning an empty value to it. Macros may be redefined, but old definitions stay around. If a macro is redefined, and the redefinition is later deleted, the first definition will take effect:

```
MAC = first        ! MAC = "first"
MAC = second       ! MAC = "second"
MAC = $(MAC) third ! MAC = "second third"
MAC =              ! MAC = "second"
MAC =              ! MAC = "first"
MAC =              ! MAC has no definition
```

A macro may be referenced in two ways:

```
 $<char>   or    $(macro-name)
```

The first way only works if the macro's name is a single character. If the macro's name is longer than one character, it must be enclosed in parenthesis. [“`$`” may be escaped by doubling it (“`$$`”.)] For example, in:

```
 G = mk.h mk1.h
 OBJS = make.obj file.obj parsedir.obj macro.obj
 BOTH = $(OBJS) $G

 make.exe : $(OBJS) $G
 make.exe : $(BOTH)
 make.exe : mk.h mk1.h make.obj file.obj
 parsedir.obj macro.obj
         echo "This is a dollar sign --> $$"
```

after macro expansion, the three dependencies will appear identical and the two “`$`”s in the last line will turn into one “`$`”.

DOS Environment variables

All MS-DOS environment variables are available within MAKE as pre-defined macro names which begin and end with “**%**”. For example, the DOS command processor (usually `command.com`) is selected by the `COMSPEC` environment variable, so you could specify it as `$(%COMSPEC%)` in your makefile. For example, you could run a DOS `.bat` file called `compile.-bat` like this:

```
 myfile.obj : myfile.c
         $(%COMSPEC%) /C compile myfile.c
```

Unix™ MAKE and this one

They are *not* the same. Do not expect Unix makefiles to work with this MAKE, even if you change the path names. There are some major differences between this version and the standard Unix™ MAKE:

1. Multiple root names are not allowed. Unix MAKE accepts lines of the form:

```
      name1 name2 : depend1 depend2
```

but this one doesn't.

2. With Unix MAKE, method lines must be preceded by a tab character. Since TLIB (and some MS-DOS editors) can convert tabs to blanks, this version of MAKE allows method lines to be preceded by either blanks or tabs.

3. There is no equivalent of double-colon (“::”).

4. There is no equivalent of `.SUFFIXES`, or the corresponding special macros.

5. This MAKE has a unique feature: it is “integrated” with TLIB and PKPAK (*a/k/a:* PKARC). That is, MAKE can check the dates of particular versions or branches within a TLIB library file, and of files which are stored in a PKPAK-style archive file (but not a PKZIP-style `.ZIP` archive - sorry).

To specify TLIB versions and branches in your MAKEFILE, simply put the version you want in brackets after the file name, like this:

```
myfile.c : myfile.c$$[4.*]
     tlib ebs myfile.c 4.*
```

You can also use a TLIB file list or snapshot version label to specify the version. For example,

```
myfile.c : myfile.c$$[@beta,myfile.c]
     tlib ebs myfile.c @beta
```

Note: Most users needn't use this feature of MAKE to ‘integrate’ it with TLIB. Instead, simply add `EqualDate Y` to your TLIB configuration file, and you can set up your MAKE dependency to check the date of the library file, since the library file date will be the same as the date of the newest version within the library file. This is much faster, since MAKE will not need to read all your TLIB library files. (For this to work, you would *not* want to configure `OldDate N`). See pp. 267 and 275.

To get MAKE to test the date of a file which is stored in compressed form within a `.ARC` archive file, just pretend that the archive file is a directory. For example,

```
fino.exe : d:\fino.arc\fino.c stdio.h
```

```
          pkunpak d:\fino fino.c
          cc fino
          link fino,fino,,lc
          del fino.c
```

PKware is obsolete. . Sorry, but we've not integrated Dyer MAKE with PKWare's current product, PKZIP.

Sample Makefile

```
# MS-DOS Make utility
# (compile with Lattice C version 2.13)
#
# Adjust these for your system:
CLIB = \lc\s\lc
COBJ = \lc\s\c
~INIT = echo on
~DEINIT = :xit

H = makedefs.h
C = make.c macro.c token.c parsedir.c file.c
FILES = $H $C osdate.asm
DOCUMENTATION = readme make.man makefile

make.obj : make.c`$ $H
        tlib e make.c
        if errorlevel 1 goto xit
        lc1 make
        lc2 make
        if errorlevel 1 pause Errors!
        erase make.c

macro.obj : macro.c $H
        lc1 macro
        lc2 macro

token.obj : token.c $H
        lc1 token
        lc2 token

parsedir.obj : parsedir.c $H
        lc1 parsedir
        lc2 parsedir

file.obj : file.c
        lc1 file
        lc2 file

osdate.obj : osdate.asm
        masm osdate;

# print the files associated with MAKE
```

```
 print :
         print make.man $(FILES) makefile

 # copy to distribution disk (on A:)
 distribution :
         copy readme a:
         copy make.man a:
         copy makefile a:
         copy make.bat a:
         copy make.c a:
         copy macro.c a:
         copy token.c a:
         copy parsedir.c a:
         copy file.c a:
         copy osdate.asm a:
         copy cmake.bat a:
         copy make.lis a:
         copy makeexe.exe a:
         copy makedefs.h a:

 # link the MAKE utility
 makeexe.exe : make.obj macro.obj token.obj
 parsedir.obj file.obj osdate.obj
         link $(COBJ) make macro token parsedir file osdate,m
akeexe,,$(CLIB)    (should be all on 1 line)
```

How Make utilites work

If you are not used to working with MAKE utilities, you may be a bit confused at this point. So let's work through a sample edit and build session in detail, using the following makefile:

```
 #makefile for "a.exe" project
 a.exe : a.obj b.obj
   rem  Link .obj files together w/ DOS linker
   link a+b,a,,
 a.obj : a.c c.h
   rem  Compile a.c w/ Lattice, producing a.obj
   lc a
 b.obj : b.c c.h d.h
   rem  Compile b.c w/ Lattice, producing b.obj
   lc b
 x.exe : x.c
   lc x
   link x,x,,
everything : a.exe x.exe
  REM  Rebuild both a.exe and x.exe
```

The first four lines are, respectively: a comment line; a "dependency line" which says that `A.EXE` depends upon both `A.OBJ` and `B.OBJ`; and then two lines of DOS commands ("build rules") to create `A.EXE` from `A.OBJ` and `B.OBJ`. On the dependency lines, the name to the left of the colon is the file to be made, called the "target," and the list of names which follow the colon is called the "prerequsite list."

here are five source files mentioned in this makefile: `A.C`, `B.C`, `C.H. D.H` and `X.C`. The five source files are used to build two programs, `A.EXE` and `X.EXE`. The compiler command "`lc`" is used to compile the C language source files, and the standard DOS linker is used to create `A.EXE` and `X.EXE` from the object files.

Assume that initially `A.EXE` and `X.EXE` are both up to date and consistent with the source files. Then you edit the file `D.H` to fix a bug; none of the other source files are modified. Then you type the command

```
make a.exe
```

First, MAKE finds the dependency line with "`A.EXE`" on the left-hand side. `A.EXE` depends upon `A.OBJ` and `B.OBJ`. The rule is that the associated command(s) must be done whenever the left hand file is either missing or is older than one or more of the right hand files, and the rule is applied recursively.

In this case, `A.EXE` is the target and `A.OBJ` & `B.OBJ` are the prerequsites. If either `A.OBJ` or `B.OBJ` is newer than `A.EXE`, then `A.EXE` must be rebuilt (the link command must be done).

Initially, `A.EXE` exists and is not newer than either of the `.OBJ` files, but since the rule is applied recursively, MAKE must first decide whether either of the `.OBJ` files must be rebuilt before it can decide about `A.EXE`.

So MAKE looks for a dependency line with `A.OBJ` ast the target (on the left); it finds one, and determines that `A.OBJ` depends upon `A.C` and `C.H`. It examines the file dates and finds that `A.OBJ` is newer than either `A.C` or `C.H`. Since neither `A.C` nor `C.H` appears as the target (left-hand side) of a dependency, MAKE concludes that `A.OBJ` does not need to be rebuilt.

Then MAKE looks for a dependency with `B.OBJ` as the target; it finds one, and determines that `B.OBJ` depends upon `B.C` and the include files `C.H` and `D.H`. It examines the file dates and finds that while `B.OBJ` is newer than either `B.C` or `C.H`, it is older than `D.H`. Thus, MAKE determines that it will be necessary to rebuild (compile) `B.OBJ`.

Before it actually emits the compile command, MAKE must first check whether any of the files which `B.OBJ` depends upon must also be rebuilt; if so, then they must be rebuilt before `B.OBJ` is rebuilt, since `B.OBJ` depends upon them.

However, none of three files which `B.OBJ` depends upon appears on the left side of a dependency. So MAKE can now emit the commands associated with rebuilding `B.OBJ`:

```
rem  Compile b.c w/ Lattice, producing b.obj
lc b
```

Since `B.OBJ` is being rebuilt, MAKE must also emit the commands to rebuild `A.EXE`, as well (because `A.EXE` depends upon `B.OBJ`):

```
rem  Link .obj files together w/ DOS linker
link a+b,a,,
```

Many MAKE utilities actually execute the specified commands immediately when they are emitted. Landon Dyer's MAKE, however, just writes them to a temporary batch file, `MAKE$$$$.BAT`, which is executed by `MAKE.BAT` as soon as the MAKE program (`MAKEEXE.EXE`) completes. This has the advantage of leaving more available RAM memory for the compilers, linkers, etc. (since the MAKE program is not in memory when they run), and it allows you to use DOS batch file commands like “`if`” and “`goto`” in your makefile.

The complete `MAKE$$$$.BAT` batch file created by our example above was:

```
rem  Compile b.c w/ Lattice, producing b.obj
lc b
rem  Link .obj files together w/ DOS linker
link a+b,a,,
```

Note that this is exactly the correct sequence of commands needed to rebuild `A.EXE` after `E.H` has been modified. Neat, eh?

If you typed “make a.exe” again, MAKE would find that both A.OBJ and B.OBJ are already up to date (newer than the source files upon which they depend), and that A.EXE is newer than either of them (and is thus also up do date). So MAKE will conclude that nothing needs to be done, and it will display the message “No changes” and quit without even creating MAKE$$$$.BAT.

Okay, you're probably thinking, but what about that strange dependency line for “everything”? That isn't even a legal file name!

The answer is that MAKE doesn't care whether it is a legal name or not; to MAKE, an illegal file name is equivalent to a file which doesn't exist.

Let's see what would have happened if we had typed the command “make everything” instead of “make a.exe”:

First, MAKE finds the dependency with “everything” on the left hand side. “everything” depends upon A.EXE and X.EXE. The rule is that the associated build rules (commands) must be done whenever the left hand file is either missing or is older than any of the right hand files, applied recursively. Since there is no file called “everything”, MAKE knows that it will need to emit the associated command(s). In this case, the only “build rule” is a REM line, but MAKE doesn't care; MAKE never even looks at the build rules; it just echoes them to MAKE$$$$.BAT. (Exception: MAKE can do macro substitutions in both build rules and dependency lines.)

However, MAKE cannot emit the rebuild command for “everything” just yet; first it must see if any of the right-hand names (prerequsites) need to be rebuilt. It checks each of them in turn, and finds that X.EXE is up to date. However, A.EXE is older than one of the files which it (indirectly) depends upon, D.H.

After determining that D.H does not, itself, need to be rebuilt, MAKE emits the commands to rebuild A.OBJ and then A.EXE:

```
rem  Compile b.c w/ Lattice, producing b.obj
lc b
rem  Link .obj files together w/ DOS linker
link a+b,a,,
```

Then, MAKE can emit the command (really just a REMark line) to rebuild “everything”, and the final MAKE$$$$.BAT file looks like this:

```
rem  Compile b.c w/ Lattice, producing b.obj
```

```
 lc b
 rem  Link .obj files together w/ DOS linker
 link a+b,a,,
 REM  Rebuild both a.exe and x.exe
```

Which is exactly what you wanted: the compile and link commands needed to rebuild just whichever `.EXE` file(s) depend upon a changed source file.

Note that if you do “make everything” again, MAKE will find that both `.EXE` files are already up to date (newer than the source files upon which they depend), so it will only try to rebuild “everything”. Thus, `MAKE$$$$.BAT` will contain only a REMark line (which does nothing), which is fine since nothing needs to be done. `MAKE$$$$.BAT` looks like this:

```
 REM  Rebuild both a.exe and x.exe
```

Easier Keyboard Input

The DOS version of TLIB's keyboard input routines were designed to be compatible with RETRIEVE (part of Personally Developed Software's UTILITIES I package, telephone 1-800-IBM-PCSW) and PCED/CED (from The Cove Software Group, P.O. Box 1072, Columbia, MD 21044, telephone 301-992-9371), as well as several similar public domain programs (DOSEDIT, NDOSEDIT, etc.). These programs provide PC-DOS with vastly improved keyboard editing facilities, including a multi-level “retrieve” key (so that you can recall previously typed commands, edit them, and re-enter them). If you use one of these programs, you'll find it to be very handy when you are updating several source files with similar or identical version definition comment lines. To retrieve the previous file's comment line, simply press the up-arrow key when TLIB's “comment line?” prompt appears.

You can also use the DOS's DOSKEY program, but it doesn't work as well because it does not separate DOS and application command histories.

Unfortunately, DOSKEY is apparently the only available command-retrieval tool for use in the “DOS box” of recent versions of Windows.

File Dates

When you use TLIB's U (update) or N (new library) command, TLIB will add the source file's creation date to the new version definition comment line. This is handy if you ever need to retrieve an old version from the library, since it can help you decide which version you need.

If you do not set the DOS date before creating or modifying your source file, TLIB will still run, but the date will be omitted from the version definition stored in the library file.

You can also configure TLIB to store just the date (not the time); see p. 266.

Note that, by default, when you extract a source file with TLIB, the source file is created with the same date and time that it had when it was stored in the library. However, you may prefer that the file be created with the current date and time, rather than the original date. One reason you may want this is to ensure that MAKE will properly reconstruct any `.OBJ` or `.EXE` files which depend upon a newly extracted source file. If you prefer this, you can change the `OLDDATE` configuration parameter; see p. 267.

Closing

We hope you find TLIB to be as useful to you as it has been to us. Although this software is sold with only a 90 day limited warranty, we want you, our customer, to be satisfied. If you have any comments, questions or complaints, please do not hesitate to call or write, even if the warranty has expired. We will do our best to help.

Our address is:

US mail: Burton Systems Software
P. O. Box 4157
Cary, NC 27519-4157 USA

UPS, etc.: Burton Systems Software
109 Black Bear Ct.
Cary, NC 27513

Our telephone and FAX numbers are:

voice: (919) 481-0149 FAX: (919) 481-3787

On the Internet you can contact us at:

email: **`support@burtonsys.com`**
URL: **`http://www.burtonsys.com/`**

Appendix A: Changes from TLIB 4

Command structure

If you upgrade to TLIB 5.5x from TLIB 4.12 or earlier, the first thing you are likely to notice is that TLIB 5 no longer uses single-letter commands (we ran out of letters).

TLIB 5.xx uses multi-character commands. This is much more flexible than the old single-character commands used in TLIB 4.12.

Each command consists of a single command character plus zero or more optional suffix characters, to modify its behavior or scope.

For example, when **U**pdating libraries, you can choose between **F**ast or regular update, between **M**inor-version-number-incremented or regular (major) number incremented, either checking-in (unlocking) or **K**eeping checked-out, etc. Thus, “UFM” (or, equivalently, “UMF”) means **U**pdate with **F**ast mode, and increment the **M**inor version number instead of the main integer version number.

With the old TLIB 4.12 single-character commands, there simply were not enough different letters available for all reasonable combinations, so some combinations were not supported.

Some DOS-only TLIB users have no need for the newly available commands, or may prefer the old commands (perhaps because they have a program or editor macro which “front-ends” TLIB and uses the old commands). To accommodate them, we have provided a mechanism by which you can configure TLIB 5.xx to look and feel more like earlier versions of TLIB. When you run TLIBCONF to set up your TLIB configuration file, it will offer to configure TLIB with old-style commands.

For more information on the command structure, including how to make TLIB 5.xx mimic TLIB 4.xx, see pp. , 59 and 330.

New & Changed Configuration Parameters

A full list of TLIB's configuration parameters can be found in Appendix D (p.   ). The following configuration parameters were new or changed in TLIB 5.0. These are listed in approximate order of importance:

Critically important, be sure to read about these: page

```
EXTENSION ext1,ext2,... 254
PATH      path(s) 257
COMMANDS  comma-delimited-list 330
```

Very important:

```
IF/ENDIF 321
LOCKING   <Y/N/B/W> 265
REPLACE   <Y/N/Q/A> 268
SET       name=unquoted-string 270
QUIET     <Y/N> 279
DETABE    <Y/N/Maybe> 255
TRACK     <Y/N/Maybe> 297
CREATETF  <Y/N> 298
PROJLEV   name 299
LEVEL     n=name d=path p=name i=names a=<Y/N/Q>
     s=<Old/New/Q/Changed> r=<Y/N> b=n c=nn f=<Y/N> 300
TREEDIRS  <Y/N> 300
DOTDOTOK  <Y/N> 303
FIXKEYWD  <Y/N> 277
READONLYB <Y/N/W> 280
DELETESRC <Y/N> 278
LOGUSER   <Y/N> 266
WORKDIR   path 304
```

Less important:

```
CMTFLAG   <number,1-80>,quoted-string 289
REFSUBDIR directory-name 305
FORCEREFR <Y/N> 306
AATTR     <Set/Preserve/Reset> 287
LIBEXT    extension 262
LOKEXT    extension 262
QUERIES   <Y/N> 269
SHEIGHT   number 284
SWIDTH    number 284
```

```
HELP      <number,1-24>,quoted-string 329
PROMPT    <number,1-24>,quoted-string 329
BANNER    <number,1-24>,quoted-string 333
CMTSUFFIX <number,1-253>,quoted-string 289
CZTRUNC   <Y/N> 291
ELSEWHERE <Y/N> 308
SLICKEPSI <Y/N/M> 292
AUTOSET   file-name 292
TRACKEXT  extension 308
```

These are the old TLIB 4.12 single-letter commands and the TLIB 5.5x equivalents, provided here for the convenience of users who are used to earlier editions of TLIB:

old		new	meaning
A	=	US	Update, Specifying version
B	=	EB	Extract, Browse mode
C	=	C,C0	Configure
E	=	E,E0	Extract and lock
F	=	UFK	Fast/freshen Update (& if locking=Y, Keep checked-out)
I	=	UD	check-in, Discarding changes (unlock)
K	=	UK	Update, Keep checked-out/locked
L	=	L,L0	List versions
M	=	UKS	Update, Specifying version, Keep checked-out
N	=	N,N0	create New library
O	=	ER	Reserve (lock without extract)
P	=	CP	Configure Path of libraries
Q	=	Q,Q0	Quit
R	=	EBS	Extract Specified version, Browse mode
S	=	NS	New lib, starting with Specified trunk version
T	=	T,T0	Test lock status
U	=	U,U0	Update, check-in
W	=	CW	Configure Who your are (ID)
X	=	ES	Extract, Specify version, check-out/lock
?	=	?,?0	Help

(note: `D`, `G`, `H`, `J`, `V`, `Y` *and* `Z` *were not valid commands in TLIB 4.12)*

Appendix B: Library File Format

For text source files (`filetype text`), the library file is also a text file, and the edit commands which are appended to it are very simple. (For binary format libraries, see p. 296.)

Since the library file is a text file, it is possible to look at it and see what changed from one version to the next. Also, though it should never be necessary, it is possible to use a text editor to change the library file itself (dangerous, but possible; one use for this is to fix erroneous comments).

There are three kinds of "edit commands" ("dot-commands") in a text-format TLIB library. They are as follows:

`.I` Insert new lines; the full edit command format is "`.I nnn`", where nnn is the number of lines to insert. The new lines appear in the library file immediately following the `.I` command line.

Note #1: if the number, `nnn`, is omitted, it is equivalent to "`1`".

Note #2: lines longer than 254 characters are counted as multiple lines.

`.C` Copy lines from previous version; the full edit command is "`.C xxx yyy`", where lines numbered `xxx` through `yyy` in the previous version of the source file are appended to the buffer which represents the new version.

Note #3: if `yyy` is omitted, it is equivalent to "`.C xxx xxx`".

Note #4: if `yyy` is less than `xxx`, it is equivalent to "`.C xxx xxx+yyy`".

`.V` Begin a version definition; a user-supplied comment appears beside the "`.V`" on the line, along with the name of the source file and the date this version was created.

Additional comment lines can follow the `.V` line, each with a "`.N`" prefix. Also, some additional information may be stored on the `.V` line, depending upon your choice of TLIB configuration parameters (see p. 266).

Let's look at a simplified example. Consider this file, called `GOODBYE.X`:

```
Dear John,
      I'm sorry, but I don't love you
anymore.  I've found someone new.  I'll
always remember the special times we've
had together.
              Love,
              Mary
```

After writing this with her trusty editor, SnazzyWrite, Mary creates a New library file for it, with the N command. Her TLIB command is:

```
TLIB N GOODBYE.X Dear John letter
```

The library file looks like this:

```
.V GOODBYE.X 14-Feb-85 Dear John letter
.I 7
Dear John,
      I'm sorry, but I don't love you
anymore.  I've found someone new.  I'll
always remember the special times we've
had together.
              Love,
              Mary
```

Every new text-format library file is in this same format: a "`.V`" command followed by a single insertion (in this case, only 7 lines).

As she is walking to the mailbox to mail her note to John, Mary spots her new beau, Mark, driving by with a beautiful blonde. Mary decides to give John a second chance. Her letter will not be wasted, however. She edits the letter to change the first line. Then she runs TLIB to Update her library file. The corrected letter looks like this:

```
Dear Mark,
      I'm sorry, but I don't love you
anymore.  I've found someone new.  I'll
always remember the special times we've
had together.
              Love,
              Mary
```

Mary's TLIB command looked like this:

```
TLIB U GOODBYE.X Dear Mark letter
```

The library file gets the following delta (version definition) appended to it to reflect the changes needed to make a Dear Mark letter out of a Dear John letter:

```
.V GOODBYE.X 15-Feb-85 Dear Mark letter
.I 1
Dear Mark,
.C 2 7
```

These two edits indicate that the new version consists of one new line followed by lines 2 through 7 of the old version.

The library file now contains two versions, and it looks like this:

```
.V GOODBYE.X 14-Feb-85 Dear John letter
.I 7
Dear John,
      I'm sorry, but I don't love you
anymore.  I've found someone new.  I'll
always remember the special times we've
had together.
              Love,
              Mary
.V GOODBYE.X 15-Feb-85 Dear Mark letter
.I 1
Dear Mark,
.C 2 7
```

Mary suspects that she will someday have use for the first version. When that time comes, she can simply run TLIB with the ES command to retrieve version 1, and avoid having to type a new letter to John.

Appendix C: Messages

We've standardized the format of TLIB's error/warning/status messages, to make it easier to spot the important ones. This change makes it more practical to start a large TLIB batch job, capture the output into a file, and later search for error messages and warnings which would indicate that something had gone wrong.

TLIB's messages, in decreasing order of importance/severity are:

`ERR:` ... TLIB internal error (you should never see this).

`ERROR:` ... Some sort of error occurred.

`Warning:` ... A mild error or other unusual event occurred.

`Note:` ... Something mildly out of the ordinary occurred.

anything else ... an informative message or interactive prompt.

CMPR, TLMERGE/DIFF3, COPYTRAK and POKETRAK also use these message formats.

If you have a “grep” utility, or an editor which supports regular expression searches, then you could find the important messages by searching for “`(ERR(OR|)|Warning|Note):`”, or something similar.

Note #1: TLIB's messages are formatted “on the fly” according to your configured `SWIDTH` (screen width). (If your computer has an IBM-compatible BIOS, then you can configure “`SWIDTH 0`” and TLIB will interrogate the system for the proper screen width.)

Note #2: To prevent you from overlooking error and warning messages, TLIB for Windows displays them in pop-up message boxes, and command-line versions of TLIB can colorize them (if you use `ANSI.SYS` or its equivalent) and/or pause to prevent them messages from scrolling off the screen. See the `COLORIZE` and `ERRORPAUS` parameters, pp. 313 And 312.

Note #3: DOS users can use Chris Dunford's nifty freeware CONCOPY utility to capture a “console log,” which you can later scan for `ERR:`, `ERROR:`, `Warning:`, and `Note:`.

You may have received CONCOPY in the “public domain and shareware” collection that came with TLIB. Otherwise, you can download it from our web site.

Note #4: OS/2 users can use a “tee” utility or the “Concurrent process buffer” of some editors to capture the output of TLIB (and other programs). We've included a `TEE.EXE` (and `tee.c`) utility with the OS/2 version of TLIB.

SlickEdit and Epsilon are the only two programmers' editors which we know of that support concurrent process buffers. Do you know of any others? Please tell us, so that we can test 'em with TLIB. Also, be sure to configure “`SlickEpsi Y`” in your TLIB configuration file if you use another editor with a concurrent process buffer. (This is unnecessary for Epsilon and for SlickEdit 2.2 or later, since TLIB can detect automatically that it is running in the concurrent process buffer of these editors, and adjust its behavior accordingly.)

Note #5: If you capture TLIB's output in a file, and process the file with a program (e.g., to find the error messages), you may wish to configure `SWIDTH 32765` to disable TLIB's on-the-fly message formatting. This ensures that each message will be on one (perhaps rather long) line, so that your program can tell where one message ends and the next begins.

Note #6: Several of TLIB's least useful but most frequently displayed status messages can be suppressed via the `-q` command line option, to make TLIB a little bit less verbose. The `-q` option (“q” for “quiet”) should be specified as the first thing on the command line after the TLIB program name. (Configuring `QUIET Y` does the same thing, except that it doesn't suppress the copyright banner.) See also p. .

Note #7: By default, when output is redirected command-line versions of TLIB will send error and warning messages to *both* “standard error” and “standard output,” so that you can still see important messages on the console even when output has been redirected into a file. This feature can be controlled via the `-e` command-line option, see p. .

Notes on a few specific messages:

```
awaiting access...
```

If you are using LAN-shared libraries, and another user has opened the shared library file (or a lock file or the journal file) for “exclusive” access, you may see this informative message. It can normally be ignored, since TLIB will retry the failed I/O operation. This situation can occur, for instance, when you List or Extract at the exact moment that someone else is doing an Update to the same library file. After a few seconds, you will usually gain access to the library file and your TLIB command will complete normally. Note that you will not see this message for a library file unless someone is doing a U (update) or N (new library) command, since the L (list) and E (extract) commands allow shared access to the TLIB libraries. Only TLIB commands which modify a file ever prevent access by other users, and then only for a brief moment.

It is, however, possible for programs other than TLIB to have “exclusive” access to a LAN-shared library file, in which case TLIB may never gain access. This could happen, for instance, if someone on a different computer were using a text editor to inspect a library file. In such cases, TLIB will eventually “time out” and display an error message. It is also possible, in a networked environment, for the source file (rather than the library file) to be opened “exclusive” by another user. This should not occur in normal operation, however, since only library files (not program source files) are commonly shared by multiple users on a LAN.

```
ERR: ...
```

The error messages beginning with “`ERR:`” are internal errors which you should never see. If you do get one of them, reboot your PC and try the operation again. If the error persists, please note what you were doing, save a copy of the file(s) you were using (both source file and library file), and notify Burton.

`ERR: Bad dot-command in library file: "`*line-from-library-file*`"`

You should never see this message. It indicates that you have a corrupted library file. The line shown in quotes was found where a "`.V`", "`.I`", or "`.C`" edit command was expected. Reboot your PC and try the operation again. If the error persists, it indicates that your library file is damaged. You can try to repair it with a text editor, or you canextract the last good version from the library file by using the ES command *from the DOS command line* (not interactively). You can list the versions with the L command to see what the last good version is (the bad version is the last one displayed). Then do "`TLIB ES` *file.ext number*" to extract the last good version. For assistance, you can call Burton tech support at (919) 481-0149.

`ERROR: could not set file date, rc=`*xxxx*

Some TLIB configuration choices require that TLIB be able to change the "last modified" date/time stamp for a file (e.g., `OLDDATE Y` and `EQUALDATE Y`). If this operation should fail, you will see this error message. The message does not indicate that a file is damaged, only that the file's date/time stamp is not what was intended.

The TOUCH program can also display this error message.

The most common cause for this problem is the use of TLIB by someone who does not have the necessary local area network file access permissions to change the date/time stamp of a file.

`Incorrect DOS version`

DOS versions of TLIB require DOS version 3.1 or higher to run. You will see this error message if you attempt to run TLIB under a really ancient version of PC-DOS or MS-DOS.

Appendix D: Configuration File Syntax

T - means TLIB uses this parameter
C - means CMPR uses it
D - means TLMERGE & DIFF3 use it

	parameter	*default*	*page*
T	`AATTR` *<set/preserve/reset>*	`Set`	287
T	`ABORT` *message*		309
T C D	`ADDCTRLZ` *<Y/N>*	`N`	276
T	`ARCCMD` *<path-of-pkpak.exe>*		328
T	`ARCEXT` *extension*	`ARC`	328
T	`ARCTEMP` *<temporary-directory>*		328
T	`AUTOBRNCH` *<Y/N/Q>*	`Query`	
T	`AUTOSET` *filename*	`AUTOSET.BAT`	292
T	`BANNER` *<1-42>*`,"`*string*`"`		333
C D	`CMPRDETAB` *<Y/N>*	`N`	339
C D	`CMPRENTAB` *<Y/N>*	`N`	339
T	`CMTEXT` *extension*	`CMT`	263
T	`CMTFLAG` *<1-80>,<quoted-string>*		289
T	`CMTSUFFIX` *<1-253>,<quoted-string>*		289
T	`COLORIZE` *<Y/N>*	`N`	313
T	`COLOROFF` *<string>*		315
T	`COMMANDS` *<comma-delimited-list>*		330
T	`CREATETF` *<Y/N>*	`N`	298
T	`CZTRUNC` *<Y/N>*	`N`	291
D	`D3COLLIDE` *<line-to-insert>*	`/* ###`...	340
D	`D3FLAG2` *<0-253>*`,"`*string*`"`		340
D	`D3FLAG3` *<0-253>*`,"`*string*`"`		340
T	`DATAPATH` *<Y/N>*	`N`	278
T	`DEFEXT` *extension*		263
T	`DELETESRC` *<Y/N>*	`N`	278
T	`DETABE` *<Y/N/Maybe>*	`Maybe`	255
C D	`DIFFLINES` *<100-16380>*	`3000`	339
T	`DOTDOTOK` *<Y/N>*	`N`	303
T	`ELSE`		323
T	`ELSEWHERE` *<Y/N>*	`N`	308
T	`ENDIF`		321
T	`ENTABU` *<Y/N>*	`N`	256

	parameter	default	page
T	`EQUALDATE` *<Y/N>*	`N`	275
T	`ERRORPAUS` *<0-3>*	`2`	312
T	`EXITPAUSE` *<Y/N>*	`N`	313
T	`EXTENSION` *ext1*`,`*ext2*`,`*...*		254
T	`FILETYPE` *<Auto/Binary/Text/Runlen/EOFtol>*	`AUTO`	294
T	`FIND1FILE` *<Y/N>*	`N`	
T	`FIXKEYWD` *<Y/N>*	`Y`	277
T	`FNAMECASE` *<U/L/A>*	`A`	309
T	`FORCEREFR` *<Y/N>*	`N`	306
T	`FORCEU` *<Y/N>*	`N`	275
T	`HELP` *<1-49>*`,"`*string*`"`		329
T	`ID` *name*		266
T	`IF` *<list-of-wildcard-specs>*		321
T	`IFF` *expression*		323
T C D	`INCLUDE` *path*		264
T	`JFILE` *<name-of-journal-file>*		190
T	`JOPTIONS` *<journal-options>*	`UOCAP` *or* `UIOAP`	190
T	`KEYFLAG` *<1-254>*`,"`*string*`"`		213
T	`LET` *name=expression*		270
T	`LEVEL` `n=`*name* `d=`*path* `p=`*name etc.*		300
T	`LIBDIRQ` *<1-54>*`,"`*string*`"`		
T	`LIBEXT` *extension*		262
T	`LOCKING` *<Y/N/B/W>*	`N`	265
T	`LOGFLAG` *<1-240>*`,"`*string*`"`		222
T	`LOGPREFIX` *<1-80>*`,"`*string*`"`		222
T	`LOGSUFFIX` *<20-253>*`,"`*string*`"`		222
T	`LOGTIME` *<Y/N>*	`Y`	266
T	`LOGUSER` *<Y/N>*	`N`	266
T	`LOGWIDTH` *<20-254>*	`79`	267
T	`LOKEXT` *extension*		262
T	`LONGNAMES` *<Y/N/M>*	`M`	77
T	`MAKEDIRS` *<Y/N>*	`N`	315
T	`MAXLINES` *<100-16380>*	`4000` or `16000`	
T	`MULTIPASS` *<Y/N>*	`Y`	335
T	`NEWLINE` *<CRLF/LF/CR>*	`CRLF`	311
T	`NT351BUG` *<Y/N>*	`N`	316
T	`NTFS35BUG` *<Y/N>*	`N`	317
T	`NUMBANNER` *<1-42>*	`0`	329
T	`NUMLIBDIR` *<1-5>*	`1`	329
T	`NUMHELP` *<1-49>*	`0`	329
T	`NUMPROMPT` *<1-42>*	`1`	329
T	`OLDDATE` *<Y/N>*	`Y`	267
T	`ONETHREAD` *<Y/N>*	`N`	292

	parameter	default	page
T	PASSSIZE *<100-16380>*	4000 or 16000	
	(this is a synonym of MAXLINES*)*		
T	PATH *<path-of-libraries>*	=	257
T	PROJLEV *name*		299
T	PROMPT *<1-42>*,"*<string>*"		329
T	QUERIES *<Y/N>*	Y	269
T	QUIET *<Y/N>*	N	279
T	READONLY *<Y/N>*	N	279
T	READONLYB *<Y/N/W>*	N	280
T	READONLYT *<Y/N/W>*	Y	320
T	REFNEWLN *<CRLF/LF/CR>*	NEWLINE	
T	REFSUBDIR *<directory-name>*		305
T	RELAXVERS *<Y/N>*	N	307
	REM *or* ! *Anything*		252
T	REPLACE *<Y/N/Q/A>*	Q	268
T	REPLROBR *<Y/N/Q/W>*	N	282
T	ROLOCKS *<Y/N>*	N	280
T	SAY *message*		309
T	SERIALNO *vvv-sssss-nn-cccccccccccc*		
T	SET *name=<unquoted-string>*		270
T	SETFTIMEW *<Y/N>*	N	274
T	SHEIGHT *<0 or 8-70>*	25	284
T	SHOWLNAME *<Y/N>*	Y	
T	SLASHCONT *<Y/N/M>*	Y	285
T	SLICKEPSI *<Y/N/Maybe>*	Maybe	292
T	SWIDTH *<0 or 40-32765>*	80	284
T	TOPRELATI *<Y/N/Maybe>*	Maybe	301
T	TOUCHSOUR *<Y/N/Modified/Revhist>*	N	
T	TOUCHU *<Y/N>*	Y	275
T	TRACK *<Y/N/Maybe>*	N	297
T	TRACKEXT *extension*	TRK	308
T	TREEDIRS *<Y/N>*	N	300
T	UNARCCMD *<path-of-pkunzip.exe>*		326
T	UPDATENEW *<Y/N>*	N	276
T	USEDUPHAN *<Y/N/Maybe>*	Maybe	285
T	USEUMBS *<Y/N>*	Y	319
T	VALIDATE *<Y/N>*	Y	285
T	WARN *message*		309
T	WORKDEPTH *nn*	0	310
T	WORKDIR *<path>*	.\ *(usually)*	304

Appendix E: TLIB Version Number Syntax

Simplified BNF Version Number Syntax

```
version          ::=  major [ : minor ] [ . branch_spec ]#
branch_spec      ::=  [ ( branch_number ) ] branch_version
major            ::=  value
minor            ::=  value
branch_number    ::=  value
branch_version   ::=  value
value            ::=  digit [ digit ]#
value            ::=  *
value            ::=  *-1
```

Where:

`::=` reads “is made up of” or “reduces to”

`[ ]` braces indicate optional clause

`[ ]#` braces with “#” means clause can be repeated 0 or more times.

Other punctuation (colon, parenthesis, period, asterisk and minus) are part of the version number syntax. However, whitespace (blanks and tabs) is never part of a version number.

Version number examples

EXAMPLE (of an unreasonably complex “tree” of version numbers)

```
 1
 ├──────────────┬───────┬────────┐
 │              │       │        │
 2             1.1    1.(2)1   1.(3)1
 ├────────┐     │                │
 │        │     │                │
2:1      2.1   1.2             1.(3)2
 ├─────┐        ├──────┐         ├──────────┬───────────┐
 │     │        │      │         │          │           │
2:2   2:1.1    1.3   1.2.1     1.(3)3    1.(3)2.1   1.(3)2.(2)1
 │     │        │
 │     │        │
 3    2:1.2    1.4
```

Trees, Trunks and Branches:

In this version number “tree,” version 1 is an ancestor of all the other versions. Versions 1, 2, 2:1, 2:2 and 3 are all “trunk versions,” so named for their resemblance to the trunk of an upside-down tree. The first trunk version (version 1) is sometimes called the “root” version. The non-trunk versions are called “branch versions.” A branch version number can be easily recognised because it contains one or more decimal points.

Major and Minor numbers:

Trunk versions are specified by either a major:minor number pair or a single integer. Specifying a single integer is equivalent to specifying a minor number of zero, so “2:0” means the same thing as “2”.

You needn't use minor numbers at all. It makes no difference to the operation of TLIB whether or not you use minor numbers; all trunk versions are handled the same way by TLIB. However, some of our customers asked for a way to distinguish between “major” revisions and “minor” changes via the version numbers, so in TLIB 5.0 we added support for minor trunk versions.

Branches:

“Branches” represent development paths which are “parallel” to the main “trunk” development sequence. Branches are typically used for bug fixes to earlier releases, or for customized versions of the module. A branch is a series of one or more branch versions descending from a particular trunk version.

The versions within a branch are numbered by “branch version numbers,” and the branches are numbered by “branch numbers.” The branch number is customarily parenthesized, and the first branch is “(1)”. The branch version number is preceded by a decimal point, and the first branch version is “.1”. The combination of branch number and branch version number, with associated punctuation, is called the branch specification. For example, “.(1)2” is a branch specification with branch number of 1 and branch version number of 2.

The complete version number for a branch version consists of the trunk version number followed by a branch specification. For example, “3.(1)2” specifies the second branch version of the first branch from trunk version 3.

In most development shops, it is rare for there to be more than one branch from a single trunk version (indeed, many shops have no need for branches of any kind), so we allow a simplified branch syntax in which the branch number is omitted and defaults to (1). Thus, “3.(1)2” could be written more simply as “3.2”.

Deep Branching:

You can also have branches off of branches. These “deep” branches are numbered in the obvious fashion, with another branch specification appended to the version number. Deep branching is very rarely needed, but, just in case, TLIB allows branches up to nine levels deep (we have never seen branches more than two levels deep in a real project). There are three examples of two-level-deep branches in the example version number tree, above: “1.2.1”, “1.(3)2.1”, and “1.(3)2.(2)1”.

More examples:

`1` Trunk version number one, usually the first version in the library file.

`1:0` Trunk version number one (same as version “1”).

`1:2` A trunk version with major number 1 and minor number 2. Read aloud as “one colon two”.

`1:02` Same as “1:0”.

`1.1` A branch version. Its predecessor is version 1. Read aloud as “one point one” or “one dot one”.

`1.01` Same as “1.1”.

`1:0.1` Same as “1.1”.

`1.(1)1` Same as “1.1”.

`1:0.(1)1` Same as “1.1”.

`5` Trunk version 5.

`6` Trunk version 6. Its predecessor is version 5 (or, perhaps, version 5:*something*).

`5.1` First version in the first branch from trunk version 5. Its predecessor is version 5.

`5.(2)1` First version in the second branch from trunk version 5. Its predecessor is also version 5.

`5.2` Second version in the first branch from trunk version 5. Its predecessor is version 5.1.

`5.0` Normally an illegal version number, but one which TLIB 5.50 tolerates and considers equivalent to specifying trunk version `5.` However, see `RELAXVERS, p. 307.`

Zero and Skipped version numbers:

Normally, TLIB does not allow you to skip version numbers. Also, TLIB normally does not allow the use of zero as a version number, neither as a trunk version number or as a branch version number. However, both of these restrictions can be circumvented with the RELAXVERS configuration parameter, p. 307.

Asterisks and Floating Version Numbers:

There are also two special non-numeric values which can be specified. An asterisk (*) means “latest”, and “*-1” means “second-to-latest.”

Thus, to select the latest trunk version, you could ask for “*:*” (this case is so common that it is normally abbreviated as just “*”). To retrieve the latest branch version from the first branch off of trunk version 5, you could select “5.*”.

Version numbers which contain asterisks are sometimes called “floating” version numbers because their effective value “floats” to the latest (or second-to-latest) version. Regular, non-floating version numbers are sometimes called “fixed” version numbers, to distinguish them from version numbers which contain asterisks.

Note that some syntactically reasonable version specifications are not actually supported by TLIB: the general rule is that the “*” or “*-1” must be the last thing specified in the version number. So, “5.*” and “3:*” are OK, but “*.5” and “*:3” are not allowed (it is unlikely that you would ever want to specify a version number that way, anyhow).

Version Labels:

Version labels are text files containing “module name / version number” pairs. A version label is extremely useful for identifying the set of modules which make up a particular release of a software product. Without a version label, if you wished to retrieve the source code for an earlier release of your program, you would have to manually select the proper version of each module, which would be a tedious and error-prone process if there were several hundred source modules!

Note: TLIB supports several ways to easily create and maintain version labels, notably the S (snapshot) command (which replaces the old TLIBSNAP program). See the index under “version label,” “snapshot,” and “version tracking file.”

Version labels which contain floating version numbers (with asterisks) are called floating version labels. Version labels which contain only fixed version numbers are called fixed version labels (or snapshots, or tracking files, depending upon the file format). Of course, there is nothing to prevent you from using hybrid version labels, which contain both fixed version numbers (for some files) and floating version numbers (for others).

With few exceptions, you can specify a version label in lieu of a version number by preceding the file name with an “at sign” (`@`).

For example, suppose that you have a version label file called “`beta.lis`” which contains the following:

```
file1.c 12.*
file2.c 5.(2)*
```

Then:

```
tlib ebs file1.c @beta.lis   means   tlib ebs file1.c 12.*
tlib us file2.c @beta.lis    means   tlib us file1.c 5.(2)*
```

Date and Time:

You can also specify trunk versions by date/time, instead of by version number or version label. In practice, however, you will probably find that you seldom use this facility. See the p. 241 for details, if you are interested.

Index