

Lahey/Fujitsu Fortran 95 User's Guide

Revision C

Copyright

Copyright © 1995-2000 Lahey Computer Systems, Inc. All rights reserved worldwide. Copyright © 1999 FUJITSU, LTD. All rights reserved. Copyright © 1986-1999 Phar Lap Software, Inc. All rights reserved. This manual is protected by federal copyright law. No part of this manual may be copied or distributed, transmitted, transcribed, stored in a retrieval system, or translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, manual, or otherwise, or disclosed to third parties.

Trademarks

Names of Lahey products are trademarks of Lahey Computer Systems, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

Disclaimer

Lahey Computer Systems, Inc. reserves the right to revise its software and publications with no obligation of Lahey Computer Systems, Inc. to notify any person or any organization of such revision. In no event shall Lahey Computer Systems, Inc. be liable for any loss of profit or any other commercial damage, including but not limited to special, consequential, or other damages.

**Lahey Computer Systems, Inc.
865 Tahoe Boulevard
P.O. Box 6091
Incline Village, NV 89450-6091
(775) 831-2500
Fax: (775) 831-8123**

<http://www.lahey.com>

**Technical Support
(775) 831-2500 (PRO version only)
support@lahey.com (all versions)**

Table of Contents

Getting Started.....	1	Mixed Language Programming.....	38
System Requirements	1	Using DLLs	38
Manual Organization	1	What Is Supported	39
Notational Conventions	2	Declaring Your Procedures.....	39
Product Registration	2	Building Fortran DLLs	40
Installing Lahey/Fujitsu Fortran 95	3	Calling DLLs from Fortran.....	41
Network Installation	3	Passing Data.....	41
Maintenance Updates	5	Delivering Applications with LF95 DLLs.....	41
Building Your First LF95 Program	5	Fortran Calling Fortran DLLs.....	41
Generating the Executable Program.....	6	C Calling Fortran DLLs	42
Running the Program	6	Fortran Calling C DLLs	42
Building Your First WiSK Program.....	7	Referencing DLL Procedures	42
Generating the Executable Program.....	7	Microsoft Visual Basic Information	45
Run the Program.....	7	Borland Delphi Information.....	47
What's Next?	8	Delphi Calling Fortran	47
Other Sources of Information	8	Fortran Calling Delphi DLLs.....	48
Developing with LF95.....	11	Examples.....	48
The Development Process	11	Calling the Windows API.....	48
How the Driver Works	12	Static Linking	49
Running LF95.....	12	OpenGL Graphics Programs	50
Filename	12	Recommended Switch Settings.....	50
Switches	13	Editing and Debugging with ED	51
Driver Configuration File (LF95.FIG)	14	Setting Up and Starting ED.....	51
Command Files.....	14	Startup.....	51
Passing Information	15	Exiting ED.....	52
Return Codes from the Driver	15	The ED Screen	52
Creating a Console-Mode Application	15	The Menu Bar	52
Creating a Windows GUI application	16	The Status Bar.....	53
Creating a WiSK Application.....	16	The Text Bar	53
Creating a 32-bit Windows DLL.....	16	Toolbars	53
Controlling Compilation.....	17	The Window Bar.....	54
Errors in Compilation.....	17	Getting Help.....	54
Compiler and Linker Switches	17	Managing Files.....	54
Linking Rules	37	Creating A File From Scratch.....	54
Fortran 90 Modules	37	Opening A File.....	55
Searching Rules.....	37	Syntax Highlighting	56
Object File Processing Rules.....	37	Navigation	56
Library Searching Rules.....	37	Previous/Next Procedure	56
		Function/Procedure List.....	56

Find	57	condition #n expr	70
Matching Parentheses and Statements	57	condition #n	70
Editing	57	oncebreak	70
Undo and Redo.....	57	regularbreak "regex"	70
Extended Characters	57	delete location	70
Blocks.....	58	delete ['file'] line	70
Coding Shortcuts	58	delete ['file'] funcname	70
Templates	58	delete *addr.....	70
Smartytype	59	delete #n.....	70
Case Conversion	59	delete.....	70
Code Completion	59	skip #n count.....	70
Compiling from ED.....	60	onstop #n cmd[;cmd2;cmd3...;cmdn] ..	71
Compiling Your Program.....	60	show break	71
Locating Errors	60	Controlling Program Execution.....	71
Changing Compiler Options	61	continue [count]	71
Debugging	61	silentcontinue [count]	71
Starting the Debugger	62	step [count]	71
Running Your Program.....	63	silentstep [count].....	71
Running a Line at a Time.....	63	steppi [count]	71
Setting Breakpoints	63	silentsteppi [count].....	71
Displaying the Values of Variables	65	next [count].....	72
Changing the Values of Variables	66	silentnext [count]	72
Reloading your Program.....	66	nexti [count].....	72
Configuration.....	66	silentnexti [count] or nin [count]	72
		until	72
		until loc	72
		until *addr.....	72
		until +/-offset.....	72
		until return.....	72
		Displaying Program Stack Information	72
Command-Line Debugging with FDB. 67		traceback [n]	72
Starting FDB.....	67	frame [#n]	73
Commands.....	67	upside [n]	73
Executing and Terminating a Program	68	downside [n]	73
run arglist.....	68	show args	73
Run	68	show locals.....	73
kill.....	68	show reg [\$r].....	73
param commandline arglist	68	show freg [\$fr]	73
param commandline	68	show regs	73
clear commandline	68	show map	73
quit.....	68	Setting and Displaying Program Variables	73
Shell Commands	68	set variable = value	73
cd dir.....	68	set *addr = value	74
pwd.....	68	set reg = value	74
Breakpoints	69		
break ['file'] line	69		
break ['file'] funcname.....	69		
break *addr	69		
break	69		

print [:F][variable]	74	breakall mdl.....	78
memprint [:FuN] addr.....	74	breakall func.....	78
Source File Display	75	show ffile.....	78
show source	75	show foft.....	79
list now	75	Communicating with fdb	79
list [next].....	75	Functions	79
list previous	75	Variables	79
list around	75	Values.....	79
list ['file'] num	75	Addresses	80
list +/-offset.....	75	Registers	80
list ['file'] top,bot	75	Names.....	80
list [func[ti]on] funcname	75	Windows Debugging with WinFDB81	
disas	76	How to Start and Terminate WinFDB	81
disas *addr1 [,*addr2]	76	Starting from the command prompt.....	81
disas funcname	76	Starting from the Windows desktop	82
Automatic Display.....	76	Starting from the ED Developer	82
screen [:F] expr.....	76	Terminating the Debugger	82
screen	76	Debugger Window	83
unscreen [#n]	76	Debugger Window	83
screenoff [#n].....	76	Debugger Menus	84
screenon [#n]	76	File Menu.....	84
show screen	76	Program Menu	84
Symbols.....	76	Debug Menu	85
show function ["regex"].....	76	Mode Menu.....	85
show variable ["regex"].....	77	Window Menu	86
Scripts.....	77	View Menu	86
alias cmd "cmd-str"	77	Help Menu	86
alias [cmd]	77	Using the Debugger.....	87
unalias [cmd]	77	Starting the Program	87
Signals	77	Setting and Deleting Breakpoints	87
signal sig action	77	Setting a Breakpoint.....	87
show signal [sig].....	77	Releasing the Breakpoint.....	88
Miscellaneous Controls	77	Running and Stopping the Program.....	89
param listsize num	77	Running the Program	89
param prompt "str"	77	Stopping the Program.....	89
param printelements num	78	Rerunning the Program	89
param prm.....	78	Displaying Debug Information	89
Files	78	Displaying Variables.....	90
show exec	78	Displaying Registers	91
param execpath [path]	78	Displaying a Traceback	91
param srcpath [path]	78	Displaying a Load Map.....	92
show source	78	Entering FDB Commands.....	92
show sources.....	78		
Fortran 95 Specific	78		

Restrictions	93	Displaying Tuning Information	120
Other Remarks.....	95	Displaying the Cost for Each Function....	121
LM Librarian.....	97	Displaying the Cost Per Line.....	121
Switches.....	97	The Calling Relationship Diagram	122
/EXTRACTALL	97	The Coverage Tool	125
/Pagesize	98	Starting and Terminating the Coverage Tool	125
/Help.....	98	Starting the Coverage Tool.....	125
Commands.....	98	Starting from the desktop icon.....	125
Add Modules.....	98	Starting from the Command prompt..	125
Delete Modules	98	Terminating the Coverage Tool.....	125
Replace Modules.....	99	Coverage Window	126
Copy Modules	99	Coverage Menus	126
Move Modules	99	File Menu.....	127
Response Files.....	99	Coverage Menu.....	127
Interactive Mode.....	100	View Menu	128
Automake	101	Window Menu	128
Introduction	101	Help Menu	128
What Does It Do?.....	101	Using the Coverage Tool	129
How Does It Do That?	101	Collecting Coverage Information	129
How Do I Set It up?	101	Storing & Merging Coverage Information ..	130
What Can Go Wrong?.....	102	Storing Coverage Information	130
Running AUTOMAKE	102	Merging Coverage Information	130
The AUTOMAKE Configuration File Editor ...	102	Displaying Coverage Information	130
The AUTOMAKE Configuration File	106	Utility Programs.....	133
Multi-Phase Compilation	111	CFG386.EXE.....	133
Automake Notes.....	112	Configuring New Switches.....	133
The Sampler Tool	115	HDRSTRIP.F90.....	134
Starting and Terminating the Sampler.....	115	PENTEST.F90	134
Starting the Sampler.....	115	SEQUNF.F90.....	134
Starting from the Sampler icon	115	TRYBLK.F90	134
Starting from the Command prompt .	116	UNFSEQ.EXE	135
Terminating the Sampler	116	WHICH.EXE	135
The Sampler Window.....	116	RSE.EXE	135
Sampler Menus	117	Programming Hints	137
File Menu	117	Efficiency Considerations.....	137
Sampler Menu	118	Side Effects	137
View Menu.....	118	File Formats	138
Window Menu.....	119	Formatted Sequential File Format	138
Help Menu.....	119	Unformatted Sequential File Format	138
Using the Sampler	119	Direct File Format	139
Collecting Tuning Information	120	Transparent File Format	139

Determine Load Image Size	139
Link Time	139
Year 2000 compliance	140
Limits of Operation.	141
Runtime Options.....	143
Command Format	143
Command Shell Variable.....	144
Execution Return Values	145
Standard Input and Output.....	145
Runtime Options	145
Description of Options	146
Shell Variables for Input/Output	151
Lahey Technical Support.....	153
Hours	153
Technical Support Services	154
How Lahey Fixes Bugs	154
Contacting Lahey	154
Information You Provide	154
Lahey Warranties	155
Return Procedure	156



Getting Started

Lahey/Fujitsu Fortran 95 (LF95) is a set of software tools for developing 32-bit Fortran applications. LF95 is a complete implementation of the Fortran 95 standard. The toolset includes a compiler, editor, linker, debugger, profiler, coverage tool, librarian, make utility, video graphics and user interface library.

LF95 includes three manuals: the *User's Guide* (this book), which describes how to use the tools; the *Language Reference*, which describes the Fortran 95 language; and the *Winteracter Starter Kit Manual*, which describes the Windows video graphics and user interface library.

System Requirements

- An 80486DX, Pentium series or compatible processor
- 24 MB of RAM (32 MB or more recommended)
- 62 MB of available hard disk space for complete installation; 55 MB for typical installation
- Windows 95, Windows 98, or Windows NT 4.0, or Windows 2000.

Manual Organization

This book is organized into eight chapters and two appendices.

- Chapter 1, *Getting Started*, identifies system requirements, describes the installation process, and takes you through the steps of building of your first program.
- Chapter 2, *Developing with LF95*, describes the development process and the driver program that controls compilation, linking, the generation of executable programs, libraries, and DLLs.
- Chapter 3, *Editing and Debugging with ED*, describes program creation and debugging using the Windows-based programming environment.

- Chapter 4, *Command-Line Debugging with FDB*, describes the command-line debugger.
- Chapter 5, *Windows Debugging with WinFDB*, describes how to automate program.
- Chapter 6, *LM Librarian*, describes command-line operation of the librarian.
- Chapter 7, *Automake*, describes how to automate program creation.
- Chapter 8, *The Sampler Tool*, describes how to profile your code to discover opportunities for execution speed optimization.
- Chapter 9, *The Coverage Tool*, describes the coverage analysis tool which can be used to determine if all portions of your code are being executed.
- Chapter 10, *Utility Programs*, describes how to use the additional utility programs.
- Appendix 11, *Programming Hints* offers suggestions about programming in Fortran on the PC with LF95.
- Appendix 12, *Runtime Options* describes options that can be added to your executable's command line to change program behavior.
- Appendix 13, *Lahey Technical Support* describes the services available from Lahey and what to do if you have trouble.

Notational Conventions

The following conventions are used throughout this manual:

Code and keystrokes are indicated by courier font.

In syntax descriptions, *[brackets]* enclose optional items.

An ellipsis, '...', following an item indicates that more items of the same form may appear.

Italics indicate text to be replaced by the programmer.

Non-italic characters in syntax descriptions are to be entered exactly as they appear.

Product Registration

To all registered LF95 users, Lahey provides free, unlimited technical support via telephone (PRO version only), fax, postal mail, and e-mail. Procedures for using Lahey Support Services are documented in Appendix 13, *Lahey Technical Support*.

To ensure that you receive technical support, product updates, newsletters, and new release announcements, please register during installation or at our website: <http://www.lahey.com>. If you move or transfer a Lahey product's ownership, please let us know.

Installing Lahey/Fujitsu Fortran 95

1. Insert the LF95 CD into your CD drive to display the Lahey/Fujitsu Fortran 95 Setup Menu.
2. If the Setup Menu does not display, run `d:\install.exe` where *d* is the drive letter of your CD drive.
3. For Windows NT and Windows 2000 users, run the installation while logged into the account which you will be using when running LF95. Administrator rights are required for installation.
4. Select 'Install Lahey/Fujitsu Fortran 95' from the Setup Menu. You'll be prompted to enter or verify your LF95 serial number. The serial number is required to install LF95 and to receive technical support. You'll also be given the choice to run Online Update as a post-installation option. This will update your product to the most recent version of LF95 from Lahey's web site.
5. *Reboot your system* (or log out and log in if using Windows NT) -- this insures that your system environment is properly configured. You are now ready to build your first program.

Network Installation

Network Administrator: The network administrator role is to install the files on a network server drive for use by users on the client systems. Running the installation with the command line arguments below will install the product files without creating icons (except the Internet folder to access Online Update) and without updating the system for use of the installed product components. It is required that you have purchased a site license from Lahey Computer Systems, Inc. or that you run monitoring software that limits the number of concurrent users of each tool in the distribution to the number of licenses purchased.

1. Enter this in the Start|Run option (do not run from the autoplay Setup Menu):

```
<d>:\Install32 netserver [ main:<n-m> ed4w:<n-e> ]
```

with these substitutions (the network pathname is the drive and directory specification):

<d> = the CD drive containing the Lahey/Fujitsu Fortran 95 CD

<n-m> = the network pathname for the compiler installation

<n-e> = the network pathname for the Lahey ED Developer installation

Note: the command line arguments surrounded by brackets are optional.

2. You will be prompted to accept the License Agreement, enter your registration information, enter/verify your installation directories, and to select which components you wish to install.
3. It is recommended that you make a batch procedure to distribute to your client users containing the command line shown below for the Network Client. Copy the file INSTALL32.EXE to a network-accessible drive for the clients to run the installation.
4. If the online documentation component is installed, then it is recommended that the Adobe Acrobat Reader install be made available for client users. This can be accomplished by copying the "ARxxxENG.EXE" installation program (where xxx is the version number) from the product CD to a network drive for the network users to run.

Network Client: Running the installation with the command line arguments below will install only those product files needed to be resident on the local system. The system files will be updated as necessary and icons will be updated as appropriate.

1. You must have the network drive mapped as a local drive letter (e.g., starting with "N:"); do not specify a UNC style name (e.g., starting with "\\"). This requirement is for proper startup of Lahey ED Developer.
2. Enter this in Start|Run (do not run from the autoplay Setup Menu):

```
<n-i>\install32 netclient [ main:<n-m> ed4w:<n-e> local:<d-l> ]
```

with these substitutions (the network pathname is the drive and directory specification):

- <n-i> = the network pathname where INSTALL32.EXE is located
- <n-m> = the network pathname where the network admin installed the compiler
- <n-e> = the network pathname where the network admin installed Lahey ED
- <d-l> = the local pathname if the default of C:\Lahey is not desired.

Note: the command line arguments surrounded by brackets are optional.

3. If "main:" is not on the command line, the client user will be prompted to accept the License Agreement, enter the network location of the product, select desired shortcuts, and to choose whether or not to have the installation program update the system environment variables.

If "main:" is contained in the command line, the client user will not be prompted for any information, and the defaults will be used.

Maintenance Updates

Maintenance updates are made available for free from Lahey's web site. They comprise bug fixes or enhancements or both for this version of LF95. The update program applies "patches" to your files to bring them up-to-date. The maintenance update version shows as a letter after the version of your compiler. This is displayed in the first line of output when you run the compiler.

To check Lahey's web site for the latest maintenance update for this version, click on Online Update in the Lahey/Fujitsu Fortran 95 Internet folder in your Programs menu, and a program will step you through it. Online Update will first perform a quick check and tell you whether you are up-to-date or if an update is available. If you choose to install the update, the necessary file patches will be downloaded and applied. You will need to be connected to the Internet to perform the check and to download the files.

Another way to get the latest maintenance update for this version is by going to this web page:

<http://www.lahey.com/patchfix.htm>

There you will find update programs you can download, as well as release notes and bug fix descriptions. Once you have downloaded an update program, you will no longer need an Internet connection. This method is preferred over Online Update by those who need to update LF95 on systems that are not connected to the Internet, or who want the ability to revert to a previous maintenance version.

Building Your First LF95 Program

LF95 is commonly referred to as a "compiler," but it is comprised of a linker, a librarian, and several other components. For this reason it is more accurately referred to as the "LF95 driver." Building and running a Fortran program with LF95 involves three basic steps:

1. Creating a source file using the Lahey ED development environment or a suitable non-formatting text editor.
2. Generating an executable program using LF95. The LF95 driver automatically *compiles* the source file(s) and *links* the resulting object file(s) with the runtime library and other libraries you specify.
3. Running the program.

The following paragraphs take you through steps two and three using the `DEMO.F90` source file included with LF95. For the sake of illustration, we will use the command line interface to invoke LF95, even though it is a windows application.

Generating the Executable Program

Compiling a source file into an object file and linking that object file with routines from the runtime library is accomplished using the LF95 . EXE driver program.

Open a system command prompt by selecting Start|Programs|Lahey-Fujitsu Fortran 95 v5.6|Command Prompt. From the command prompt, build the demo program by changing to the directory where DEMO . F90 in LF95's EXAMPLES directory by default, and entering

```
LF95 demo
```

This causes the compiler to read the source file DEMO . F90 (the extension . F90 is assumed by default) and compile it into the object file DEMO . OBJ. Once DEMO . OBJ is created, LF95 invokes the linker to combine necessary routines from the runtime library and produce the executable program, DEMO . EXE.

Running the Program

To run the program, type its name at the command prompt:

```
demo
```

and press Enter. The DEMO program begins and a screen similar to the following screen displays:

```
Lahey LF90 Compiler
-----
installation test and demonstration program

Copyright (c) 1995
Lahey Computer Systems, Inc.

-----
Test/Action List:
-----
1 - factorials
2 - Fahrenheit to Celsius conversion
3 - Carmichael numbers
4 - Ramanujan's series
5 - Stirling numbers of the 2nd kind
6 - chi-square quantiles
7 - Pythagorean triplets
8 - date_and_time, and other system calls
0 - <stop this program>

Please select an option by entering the
associated number followed by <return>.
```

You've successfully built and run the Lahey demonstration program.

Building Your First WiSK Program

LF95 comes bundled with a graphics library called WiSK (the *Winteracter* Starter Kit) which is derived from the full *Winteracter* library created by Interactive Software Services, Ltd. *Winteracter* is a Win32 and Fortran 90 dedicated user-interface and graphics development tool that allows Fortran programmers to incorporate dialogs, menus, presentation graphics, and other windows features into their applications. Building and running an LF95 Windows program with WiSK is accomplished as follows:

1. Create a user interface using the DialogEd and MenuEd design tools (fully documented in the *Winteracter* Starter Kit Manual). These tools will generate a Windows resource (.rc) file.
2. Create a Fortran source file using the Lahey ED for Windows editor or any other ASCII text editor. Parameters defined in the resource file and used in the Fortran source must be declared in the Fortran source as well (see the *Winteracter* Starter Kit Manual for a detailed explanation).
3. Generate an executable program using the LF95 driver. Specify the `-wisk` switch and include your Fortran source(s) and resource file on the command line.
4. Run the program.

The following paragraphs take you through steps three and four using the WISKDEMO example included with LF95.

Generating the Executable Program

To create the executable program, first locate the file WISKDEMO.F90 in the EXAMPLES directory, then enter

```
LF95 wiskdemo.F90 resource.rc -wisk
```

WISKDEMO.OBJ and RESOURCE.RES are created. LF95.EXE then automatically links in the appropriate libraries to produce a 32-bit Windows executable program, WISKDEMO.EXE.

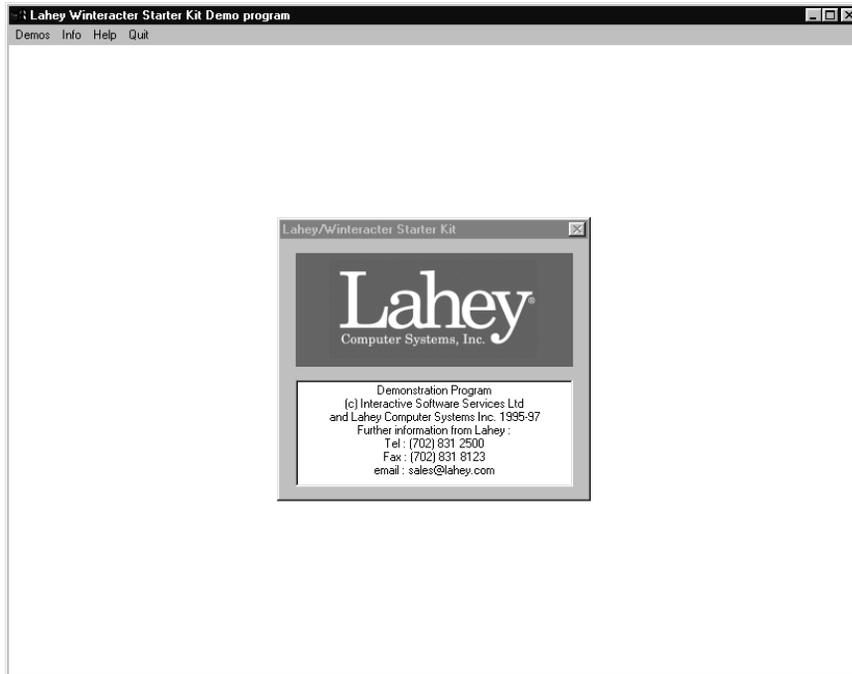
Now that you have mastered the command line, you are ready to try building programs from within the Lahey ED development environment. We recommend that you first read "*Editing and Debugging with ED*" on page 51.

Run the Program

To run the program, enter:

```
wiskdemo
```

Alternately, click ED's OS Program: Run button. The program begins and a window similar to the following displays:



You've successfully created the *WiSK* demo program for Windows.

What's Next?

For a more complete description of the development process and instructions for using Lahey/Fujitsu Fortran 95, please turn to Chapter 2, *Developing with LF95*.

Before continuing, however, please read the files `readme.txt` and `errata.txt`. These contain important last-minute information and changes to the documentation.

Other Sources of Information

Files

<code>README.TXT</code>	last-minute information
<code>README_API.TXT</code>	Windows API programming information
<code>README_C.TXT</code>	C-interface documentation
<code>README_ASSEMBLY.TXT</code>	assembly-interface documentation
<code>README_WISK.TXT</code>	last-minute <i>WiSK</i> information

README_COMPATIBLE.TXT compatible products directory
README_F90GL.TXT F90GL (Fortran bindings to OpenGL) information
README_F90SQL.TXT f90SQL-Lite information
README_SERVICE_ROUTINES.TXT POSIX and other service routines
FILELIST.TXT description of all files distributed with LF95
ERRATA.TXT changes that were made after the manuals went to press
LINKERR.TXT linker error messages

Manuals (supplied both on-line and in hard copy)

Lahey/Fujitsu Fortran 95 Language Reference
Winteracter Starter Kit

Manuals (supplied on-line only)

C Compiler User's Guide (if selected at installation time)
SSL2 User's Guide
SSL2 Extended Capabilities User's Guide
SSL2 Extended Capabilities User's Guide II
Visual Analyzer User's Guide

Help Files

WiSK Help
f90SQL-Lite Help

Newsletters

The Lahey Fortran *Source* newsletter

Lahey Web Page

<http://www.lahey.com>

2

Developing with LF95

This chapter describes how to use Lahey/Fujitsu Fortran 95. It presents an overview of the development process and describes how to build Fortran applications using the LF95 driver. The driver controls compilation, linking, and the production of executable programs and dynamic link libraries (DLLs).

The Development Process

Developing applications with LF95 involves the following tools:

Editor. Use the Lahey ED development environment to create or modify Fortran source files. The driver can be run from within the editor. Compiler error messages are automatically keyed to lines of source code. ED also integrates debugging facilities for Windows applications. See Chapter 3, *Editing and Debugging with ED*, for instructions on using Lahey ED.

Library Manager. Use the library manager to create, change, and list the contents of object libraries. See Chapter 6, *LM Librarian*, for instructions on how to use the library manager.

Automake. Use the Automake utility to automate program creation. This is especially useful if your program consists of multiple files. See Chapter 7, *Automake*, for instructions on how to use Automake.

Debuggers. For Windows console and GUI applications use FDB or WinFDB to debug your code (See Chapter 4, *Command-Line Debugging with FDB* and Chapter 5, *Windows Debugging with WinFDB*).

Driver. Use the driver (LF95.EXE) to control the creation of object files, libraries, executable programs, and DLLs. LF95.EXE is often referred to as the compiler, but it is actually a driver that invokes the compiler, linker, and other components used to create executables, libraries, and other products.

The remainder of this chapter focuses on the driver and the processes it controls.

How the Driver Works

The driver (LF95 . EXE) controls the two main processes—compilation and linking—used to create an executable program. Two supplemental processes, creating import libraries and processing Windows resources, are sometimes used depending on whether you are creating a DLL or a 32-bit Windows program. These processes are performed by the following programs under control of the driver:

Compiler. The compiler compiles source files into object files and creates files required for using Fortran 90 modules and files needed by the linker for creating DLLs.

Library Manager. LM . EXE is the library manager. It can be invoked from the driver or from the command prompt to create or change static libraries.

Linker. 386LINK . EXE is the linker. The linker combines object files and libraries into a single executable program or dynamic link library. The linker also adds Windows resources, like icons and cursors, into Windows executables.

Import library manager. Import library managers are provided with various 32-bit Windows user interface tools. From definition files output by the compiler, an import library manager creates import libraries for use with LF95 dynamic link libraries (DLLs).

Resource Compiler. RC . EXE is the resource compiler. It converts Windows resource files (.RC files) to .RES files. .RES files are converted by RES2OBJ . EXE into object files.

Running LF95

To run the driver, type LF95 followed by a list of one or more file names and optional command-line switches:

LF95 filenames [switches]

The driver searches for the various tools (the compiler, library manager, linker, import library manager, and resource compiler) first in the directory the driver is located and then, if not found, on the DOS path. The command line switches are discussed later in this chapter.

Filenames

Depending on the extension(s) of the filename(s) specified, the driver will invoke the necessary tools. The extensions .F95, .F90, .FOR, and .F, for example, cause the compiler to be invoked. The extension .OBJ causes the linker to be invoked; the extension .RC causes the resource compiler, RES2OBJ, and the linker to be invoked. .RES causes RES2OBJ and the linker to be invoked.

Filenames containing spaces must be enclosed in quotes.

Note: the extension `.MOD` is reserved for compiler-generated module files. Do not use this extension for your Fortran source files.

Source Filenames

One or more source filenames may be specified, either by name or using the DOS wildcards `*` and `?`. Filenames must be separated by a space.

Example

```
LF95 *.f90
```

If the files `ONE.F90`, `TWO.F90`, and `THREE.FOR` were in the current directory, `ONE.F90` and `TWO.F90` would be compiled and linked together, and the stub-bound executable file, `ONE.EXE`, would be created because the driver found `ONE.F90` before `TWO.F90` in the current directory. `THREE.FOR` would not be compiled because its extension does not match the extension specified on the `LF95` command line.

Source filenames are specified as a complete file name or can be given without an extension, in which case `LF95` supplies the default extension `.F90`. In the absence of a switch specifying otherwise:

- `.F90` specifies interpretation as Fortran 90 free source form.

- `.FOR` and `.F` specify interpretation as Fortran 90 fixed source form.

If files with both the `.FOR` or `.F` and `.F90` appear on the same command line, then all are assumed to use the source form the driver assumes for the last file specified.

The `-fix` and `-nfix` compiler switches can be used to control the assumed extension and override the interpretation specified by the extension. see “*-[N]FIX*” on page 23

Object Filenames

The default name for an object file is the same as the source file name. If a path is specified for the source filename, the same path will be used for the object file name. If no path is specified, the current directory will be used.

Output Filenames

The default name for the executable file or dynamic link library produced by the driver is based on the first source or object name encountered on the command line. This may be overridden by specifying the `-OUT` switch with a new name. see “*-OUT filename*” on page 29
The default extension for executable files is `.EXE`. The default extension for dynamic link libraries is `.DLL`.

Switches

The driver recognizes one or more letters preceded by a hyphen (`-`) as a command-line switch. You may not combine switches after a hyphen: for example, `-x` and `-y` may not be entered as `-xy`.

Some switches take arguments in the form of filenames, strings, letters, or numbers. You must enter a space between the option and its argument(s).

Example

```
-i incdir
```

If an unknown switch is detected, the entire text from the beginning of the unknown switch to the beginning of the next switch or end of the command line is passed to the linker.

Conflicts Between Switches

Command line switches are processed from left to right. If conflicting switches are specified, the last one specified takes precedence. For example, if the command line contained LF95 foo -g -ng, the -ng switch would be used.

To display the LF95 version number and a summary of valid command-line options, type LF95 without any command-line switches or filenames.

Driver Configuration File (LF95.FIG)

In addition to specifying switches on the command line, you may specify a default set of switches in the LF95.FIG file. When the driver is invoked, the switches in the LF95.FIG file are processed before those on the command line. Command-line switches override those in the LF95.FIG file. The driver searches for LF95.FIG first in the current directory and then, if not found, in the directory in which the driver is located.

Command Files

If you have too many switches and files to fit on the command line, you can place them in a command file. Enter LF95 command line arguments in a command file in exactly the same manner as on the command line. Command files may have as many lines as needed. Lines beginning with an initial # are comments.

To process a command file, preface the name of the file with an @ character. When LF95 encounters a filename that begins with @ on the command line, it opens the file and processes the commands in it.

Example

```
LF95 @mycmds
```

In this example, LF95 reads its commands from the file mycmds.

Command files may be used both with other command-line switches and other command files. Multiple command files are processed left to right in the order they are encountered.

Passing Information

The LF95 driver uses temporary files for sending information between the driver and processes it controls. These files are automatically created using random names and are deleted.

Return Codes from the Driver

When the LF95 driver receives a failure return code, it aborts the build process. The driver will return an error code depending on the success of the invoked tools. These return codes are listed below:

Table 1: Driver Return Codes

Code	Condition
0	Successful compilation and link
1	Compiler fatal error
2	Library Manager error
3	Linker error
4	Driver error
5	Help requested
8	RES2OBJ error
9	Resource compiler error

Creating a Console-Mode Application

To create a Windows console-mode executable that will run on Windows 95, Windows 98, or Windows NT, no switches need be specified.

Example

```
LF95 MYPROG.F90
```

Creating a Windows GUI application

To create a Windows GUI application, either with a third-party package (such as Winteracter, GINO, or RealWin) or by calling the Windows API's directly, specify the `-win` switch. To call the Windows API's directly, you must also specify the `-ml winapi` switch (see "*ML target*" on page 28 and "*Calling the Windows API*" on page 48 for more information).

Example

```
LF95 MYPROG.F90 -win
```

Creating a WiSK Application

To create a 32-bit Windows program using routines from the WiSK library, specify the `-wisk` switch along with the name of a resource file created with DialogEd and MenuEd.

Example

```
LF95 myprog.f90 myrc.rc -wisk
```

In this example, the source file `MYPROG.F90` contains calls to the WiSK library and `MYRC.RC` contains resource definitions created by MenuEd and DialogEd. The following takes place:

1. `MYPROG.F90` is compiled to create `MYPROG.OBJ`.
2. `MYRC.RC` is compiled to create `MYRC.RES`.
3. `MYRC.RES` is processed by `RES2OBJ` to create `MYRC.OBJ`.
4. `MYPROG.OBJ` and `MYRC.OBJ` are automatically linked with the LF95 runtime library and `WISK.LIB`, to create `MYPROG.EXE`, a 32-bit Windows executable.

Creating a 32-bit Windows DLL

To create a 32-bit Windows DLL, use the `-dll` switch.

Example

```
LF95 myprog.f90 -dll -win -ml msvc
```

In this example, the source file `MYPROG.F90` contains routines with `DLL_EXPORT` statements. The following takes place:

1. `MYPROG.F90` is compiled to create `MYPROG.OBJ`.
2. `MYPROG.OBJ` is automatically linked with the LF95 runtime library to create `MYPROG.DLL` and `MYPROG.LIB`, the corresponding import library. Calling conventions in this case are those expected by Microsoft Visual C/C++.

For more information on DLLs, see "*Using DLLs*" on page 38.

Controlling Compilation

During the compilation phase, the driver submits specified source files to the compiler for compilation and optimization. If the `-c`, compile only, switch is specified, processing will stop after the compiler runs and modules are created (if necessary). See `"-[N]C"` on page 19. Otherwise, processing continues with linking and possibly import library creation.

Errors in Compilation

If the compiler encounters errors or questionable code, you may receive any of the following types of diagnostic messages (a letter precedes each message, indicating its severity):

U:Unrecoverable error messages indicate it is not practical to continue compilation.

S:Serious error messages indicate the compilation will continue, but no object file will be generated.

W:Warning messages indicate probable programming errors that are not serious enough to prevent execution. Can be suppressed with the `-nw` or `-swm` switch.

I:Informational messages suggest possible areas for improvement in your code and give details of optimizations performed by the compiler. These are normally suppressed, but can be seen by specifying the `-info` switch (see `"-[N]INFO"` on page 25).

If no unrecoverable or serious errors are detected by the compiler, the `DOS ERRORLEVEL` is set to zero (see *"Return Codes from the Driver"* on page 15). Unrecoverable or serious errors detected by the compiler (improper syntax, for example) terminate the build process. An object file is not created.

Compiler and Linker Switches

You can control compilation and linking by using any of the following option switches. These switches are not case sensitive. Some switches apply only to the compilation phase, others to the linking phase, and still others (`-g`, `-win`, and `-wisk`) to both phases; this is indicated next to the name of the switch. If compilation and linking are performed separately (i.e., in separate command lines), then switches that apply to both phases must be included in each command line.

Compiling and linking can be broken into separate steps using the `-c` switch. Unless the `-c` switch is specified, the LF95 driver will attempt to link and create an executable after the compilation phase completes. Specifying `-c` anywhere in the command line will cause the link phase to be abandoned and all linker switches to be ignored.

Note also that linker switches may be abbreviated as indicated by the uppercase characters in the switch name. For example, the `-LIBPath` switch can be specified as either `-libpath` or `-libp`. Some linker switches require a number as an argument. By default, all numbers are assumed to be decimal numbers. A different radix can be specified by appending a radix specifier to the number. The following table lists the bases and their radix specifiers:

Table 2: Radix Specifiers

Base	Radix Specifier	Example of 32 in base
2	B or b	10000b
8	Q or q	40q
10	none	32
16	H or h	20h

The underscore character ('_') can be used in numbers to make them more readable: `80000000h` is the same as `8000_0000h`.

-[N]AP**Arithmetic Precision**

Compile only. Default: `-nap`

Specify `-ap` to guarantee the consistency of `REAL` and `COMPLEX` calculations, regardless of optimization level; user variables are not assigned to registers. Consider the following example:

Example

```

X = S - T
2 Y = X - U
...
3 Y = X - U

```

By default (`-nap`), during compilation of statement 2, the compiler recognizes the value `X` is already in a register and does not cause the value to be reloaded from memory. At statement 3, the value `X` may or may not already be in a register, and so the value may or may not be reloaded accordingly. Because the precision of the datum is greater in a register than in memory, a difference in precision at statements 2 and 3 may occur.

Specify `-ap` to choose the memory reference for non-`INTEGER` operands; that is, registers are reloaded. `-ap` must be specified when testing for the equality of randomly-generated values.

The default, `-nap`, allows the compiler to take advantage of the current values in registers, with possibly greater accuracy in low-order bits.

Specifying `-ap` will usually generate slower executables.

-[/NO]BANNER

Linker Banner

Link only. Default: `-banner`

`-banner` displays a 386|LINK copyright message with the 386|LINK version and serial number. `-nobanner` suppresses the 386|LINK copyright message.

-BLOCK *blocksize*

Default blocksize

Compile only. Default: 8192 bytes

Specify `-block` to default to a specific blocksize on OPEN statements. See “*blocksize*” in the LF95 Language Reference. *blocksize* must be a decimal INTEGER constant. Specifying an optimal *blocksize* can make an enormous improvement in the speed of your executable. The program TRYBLOCK.F90 in the SRC directory demonstrates how changing blocksize can affect execution speed. Some experimentation with *blocksize* in your program is usually necessary to determine the optimal value.

-[/N]C

Suppress Linking

Compile only. Default: `-nc`

Specify `-c` to create object (.OBJ), and, if necessary, module (.MOD) files without creating an executable. This is especially useful in makefiles (see “*Automake*” on page 101), where it is not always desirable to perform the entire build process with one invocation of the driver.

-[/N]CHK

Checking

Compile only. Default: `-nchk`

Specify `-chk` to generate a fatal runtime error message when substring and array subscripts are out of range, when non-common variables are accessed before they are initialized, when array expression shapes do not match, and when procedure arguments do not match in type, attributes, size, or shape.

Syntax

```
-[n]chk [( [a] [ , e] [ , s] [ , u] [ , x] )]
```

Note: Commas are optional, but are recommended for readability.

Table 3: -chk Arguments

Diagnostic Checking Class	Switch Argument
Arguments	a
Array Expression Shape	e
Subscripts	s
Undefined variables	u
Increased (extra)	x

Specifying `-chk` with no arguments is equivalent to specifying `-chk (a,e,s,u)`. Specify `-chk` with any combination of a, e, s, u and x to activate the specified diagnostic checking class.

Specification of the argument x must be used for compilation of all files of the program, or incorrect results may occur. Do not use with 3rd party compiled modules, objects, or libraries. Specifically, the x argument must be used to compile all USED modules and to compile program units which set values within COMMONs. Specifying the argument x will force undefined variables checking (u), and will increase the level of checking performed by any other specified arguments.

Specifying `-chk` adds to the size of a program and causes it to run more slowly, sometimes as much as an order of magnitude. It forces `-trace` and removes optimization by forcing `-o0`.

Example

```
LF95 myprog -chk (a,x)
```

instructs the compiler to activate increased runtime argument checking and increased undefined variables checking.

The `-chk` switch will not check bounds in the following conditions:

- The referenced expression has the POINTER attribute or is a structure one or more of whose structure components has the POINTER attribute.
- The referenced expression is an assumed-shape array.
- The referenced expression is an array section with vector subscript.
- The referenced variable is a dummy argument corresponding to an actual argument that is an array section.
- The referenced expression is in a masked array assignment.
- The referenced expression is in a FORALL statements and constructs.
- The referenced expression has the PARAMETER attribute.
- The parent string is a scalar constant.

-/N/CHKGLOBAL

Global Checking

Compile only. Default: `-nchkglobal`

Specify `-chkglobal` to generate compiler error messages for inter-program-unit diagnostics, and to perform full compile-time and runtime checking.

The global checking will only be performed on the source which is compiled within one invocation of the compiler (the command line). For example, the checking will not occur on a USEd module which is not compiled at the same time as the source containing the USE statement, nor will the checking occur on object files or libraries specified on the command line.

Because specifying `-chkglobal` forces `-chk (x)`, specification of `-chkglobal` must be used for compilation of all files of the program, or incorrect results may occur. Do not use with 3rd-party-compiled modules, objects, or libraries. See the description of `-chk` for more information.

Global checking diagnostics will not be published in the listing file. Specifying `-chkglobal` adds to the size of a program and causes it to run more slowly, sometimes as much as an order of magnitude. It forces `-chk (a,e,s,u,x)`, `-trace`, and removes optimization by forcing `-o0`.

-/N/CO

Compiler Options

Compile *and* link. Default: `-co`

Specify `-co` to display current settings of compiler options; specify `-nco` to suppress them.

-/N/COVER

Coverage Information

Compile *and* link. Default: `-ncover`

Specify `-cover` to generate information for use by the coverage tool (see Chapter 9, *The Coverage Tool*). This switch is required to run the coverage tool if a separate link is performed.

-/N/DAL

Deallocate Allocatables

Compile only. Default: `-dal`

Specify `-dal` to deallocate allocated arrays that do not appear in DEALLOCATE or SAVE statements when a RETURN, STOP, or END statement is encountered in the program unit containing the allocatable array. Note that `-ndal` will suppress automatic deallocation, even for Fortran 95 files (automatic deallocation is standard behavior in Fortran 95).

-/N/DBL

Double

Compile only. Default: `-ndbl`

Specify `-dbl` to extend all single-precision REAL and single-precision COMPLEX variables, arrays, constants, and functions to REAL (KIND=8) and COMPLEX (KIND=8) respectively. If you use `-dbl`, all source files (including modules) in a program should be compiled with `-dbl`. Specifying `-dbl` will usually result in somewhat slower executables.

-/N/DLL

Dynamic Link Library

Link only. Default: `-ndl1`

Specify `-dll`, `-ml`, and `-win` to create a 32-bit Windows dynamic link library (for more information, see "Using DLLs" on page 38).

-/N/F90SQL

Use f90SQL Lite

Compile and link. Default: `-nf90sql`

Specify `-f90sql` to create an application using f90SQL Lite.

-/N/F95

Fortran 95 Conformance

Compile only. Default: `-nf95`

Specify `-f95` to generate warnings when the compiler encounters non-standard Fortran 95 code.

Note that `-nf95` allows any intrinsic data type to be equivalenced to any other.

-FILE filename

Filename

Compile *and* link. Default: not present

Precede the name of a file with `-file` to ensure the driver will interpret the filename as the name of a file and not an argument to a switch.

Example

On the following command line, `bill.f90` is correctly interpreted as a source file:

```
LF95 -checksum -file bill.f90
```

On this next command line, `bill.f90` is not recognized as a source file. The driver passes the unrecognized switch, `-checksum`, to the linker and assumes the following string, "`bill.f90`", is an argument to the `-checksum` switch.

```
LF95 -checksum bill.f90
```

On this last command line, `-file` is not necessary. The order of driver arguments allows unambiguous interpretation:

```
LF95 bill.f90 -checksum
```

-/N/FIX

Fixed Source Form

Compile only. Default: `-nfix` for `.f90` and `.f95` files; `-fix` for `.for` or `.f` files

Specify `-fix` to instruct the compiler to interpret source files as Fortran 90 fixed source form. `-nfix` instructs the compiler to interpret source files as Fortran 90 free source form.

Example

```
LF95 @bob.rsp bill.f90
```

If the command file `BOB.RSP` contains `-fix`, `BILL.F90` will be interpreted as fixed source form even though it has the free source form extension `.F90`.

LF95 assumes a default file extension of `.f90`. Specifying `-fix` causes LF95 to assume a default file extension of `.for`.

All source files compiled at the same time must be fixed or free. LF95 doesn't compile files (including `INCLUDE` files) containing both fixed and free source form.

-/N/G

Debug

Compile *and* link. Default: `-ng`

Specify `-g` to instruct the compiler to generate an expanded symbol table and other information for the debugger. `-g` automatically overrides any optimization switch and forces `-o0`, no optimizations, so your executable will run more slowly than if one of the higher optimization levels were used. `-g` is required to use the debugger. Supplemental debug information is stored in a file having the same name as the executable file with extension `.YDG`. If the following error message appears during linking

```
fdwmerge:[error] Terminated abnormally. (signal 11)
```

It means that the `.YDG` file was not created (contact Technical Support if this happens).

This switch is required to debug if a separate link is performed.

-I path

Include Path

Compile only. Default: current directory

Specify `-i path` to instruct the compiler to search the specified path(s) for Fortran `INCLUDE` files after searching the current directory. Separate multiple search paths with a semicolon, no spaces.

Example

```
LF95 demo -i ..\project2\includes;..\project3\includes
```

In this example, the compiler first searches the current directory, then searches `..\project2\includes` and finally `..\project3\includes` for `INCLUDE` files specified in the source file `DEMO.F90`

-IMPLIB

Specify DLL Library

Link only. Default: not specified.

The `-implib` switch specifies the name of the dynamic link library from which a program has called functions listed in one or more subsequent `-import` switches. The library specified in the `-implib` switch is valid for all modules specified in any number of subsequent `-import` switches, until the specification of another `-implib` switch.

Syntax

```
-IMPLIB libname
```

where *libname* is the name and path of the DLL library from which to import functions using subsequent `-import` statements.

Example

```
LF95 main -implib mydll -import myfunc1,myfunc2
LF95 main -implib A -import A1 -implib B -import B1,B2,B3
```

-IMPORT

Import a DLL Function

Link only. Default: not specified.

Syntax

```
-IMPORT dllname.funcname
```

or

```
-IMPORT anyfunc=dllname.funcname
```

where *dllname* is the DLL library from which to import the function, *funcname* is the name of the function being imported, and *anyfunc* is an alias for the name of the actual function (the alias is useful for handling differences in naming conventions). The name of the DLL library can be omitted if it was specified previously with the `-implib` switch.

Example

```
LF95 main -import mathlib.getnum mathlib.tan
LF95 main -import _getnum@12=mathlib.getnum
LF95 main -implib mathlib -import _getnum@12=getnum
```

-(N)IN

Implicit None

Compile only. Default: `-nin`

Specifying `-in` is equivalent to including an IMPLICIT NONE statement in each program unit of your source file: no implicit typing is in effect over the source file.

When `-nin` is specified, standard implicit typing rules are in effect.

-[N]INFO

Display Informational Messages

Compile only. Default: `-ninfo`

Specify `-info` to display informational messages at compile time. Informational messages include such things as the level of loop unrolling performed, variables declared but never used, divisions changed to multiplication by reciprocal, etc.

-[N]LI

Lahey Intrinsic Procedures

Compile and link. Default: `-li`

Specify `-nli` to avoid recognizing non-standard Lahey intrinsic procedures.

-Lib filename

Library

Link only. Default: none

The `-Lib` switch specifies one or more library files. The names of the library files immediately follow the switch, separated by either spaces or commas. If no filename extension is specified for a library file, the linker assumes the extension, `.LIB`.

The `-Lib` switch may be used multiple times in a single linker command string. The linker builds a list of the library files and processes them in the order they were specified on the command line.

To create and maintain libraries of commonly-used functions, use the LM library manager. See Chapter 6, *LM Librarian*.

Syntax

```
-Lib lib1[,lib2 ...]
```

lib1 and *lib2* are one or more library files.

Example

```
LF95 hello.obj -lib mylib
```

-LIBPath path

Library Path

Link only. Default: current directory.

The `-LIBPath` switch allows specification of one or more directories to be searched for libraries. Note that all necessary library files must still be called out in the command line.

Syntax

```
-LIBPath dir1[,dir2 ...]
```

dir1 and *dir2* are one or more directories to be searched.

Example

```
LF95 main.obj -libpath d:\mylibs -lib mine.lib -pack
LF95 main.obj -libp d:\mylibs,e:\yourlibs -lib mine,yours
```

Directory names specified for `-LIBPath` must not end with a “\” delimiter. The linker will affix the directory delimiter to the file name being sought.

-/N/LONG

Long Integers

Compile only. Default: `-nlong`

Specify `-long` to extend all default INTEGER variables, arrays, constants, and functions to INTEGER (KIND=8). If you use `-long`, all source files (including modules) in a program should be compiled with `-long`.

-/N/LST

Listing

Compile only. Default: `-nlst`

Specify `-lst` to generate a listing file that contains the source program, compiler options, date and time of compilation, and any compiler diagnostics. The compiler outputs one listing file for each source file specified. By default, listing file names consist of the root of the source file name plus the extension `.lst`.

Syntax

```
-[n]lst [(spec=sval], spec = sval] ... )]
```

Where:

`spec` is `f` for the listing file name, or `i` to include INCLUDE files.

For `f=sval`, the listing file name, `sval` specifies the listing file name to use instead of the default. If a file with this name already exists, it is overwritten. If the file can't be overwritten, the compiler aborts. If the user specifies a listing file name and more than one source file (possibly using wild cards) then the driver diagnoses the error and aborts.

For `i=sval`, `sval` is one of the characters of the set `[YyNn]`, where `Y` and `y` indicate that include files should be included in the listing and `N` and `n` indicate that they should not. By default, include files are not included in the listing.

Example

```
LF95 myprog -lst (i=y)
```

creates the listing file `myprog.lst`, which lists primary and included source. Note that `-xref` overrides `-lst`.

See also

`-/N/XREF`

-[NO]Map filename

Map File

Link only. Default: `-map`

The linker map file is a text file describing the output load image. The map file contains the following information:

- command switches specified when the program was linked,
- names of the input object files,
- a list of the segments comprising the program, and
- a list of the public symbols in the program.

By default, the linker produces a map file each time a program is linked. The default name of the map file is the name of the output file, with its extension changed to `.MAP`. Any path information specifying a directory where the output file is to be placed also applies to the map file.

The `-Map` switch renames or relocates the map file. The switch takes a single argument, which is the path and name of the map file to be produced. If no file name extension is specified, then a default of `.MAP` is assumed. If no path information is specified in the map file name, then it is placed in the current directory.

The linker can be prevented from producing a map file with the `-NOMap` switch. The switch takes no arguments. The `-NOMap` switch is useful to make the linker run faster, since no time is spent writing the map file. The switch is also a good way to save disk space, because map files can become quite large.

Syntax

`-Map filename`

Example

```
LF95 moe.obj larry.obj curly.obj -m stooges
LF95 hello.obj -nom
```

-MAPNames nchars

Mapfile Name Length

Link only. Default: `-mapnames 12`

The `-MAPNames` switch controls the length of global symbol names displayed in the map file. By default, segment, group, class, module, and public symbol names are truncated to 12 characters in the map file. The switch takes a numeric constant argument which increases the length of global symbols in the map file to the specified number of characters.

Increasing the symbol name length may cause the default maximum line width of 80 characters to be exceeded. If this occurs, the linker prints less information about segments and public symbols. This loss of information can be prevented by using the `-MAPWidth` switch.

Syntax

`-MAPNames nchars`

nchars is the length of global symbols in the map file, expressed as number of characters.

Example

```
LF95 hello.obj -mapn 30
```

-MAPWidth *nchars*

Mapfile Line Width

Link only. Default: `-mapwidth 80`

The `-MAPwidth` switch controls the maximum line width in the program map file. The switch takes a numeric constant argument which is the new maximum width for lines in the map file.

Syntax

```
-MAPwidth nchars
```

nchars is the maximum line width in the map file, expressed as number of characters.

Example

```
LF95 hello.obj -mapn 30 -mapw 120
```

-/N/MAXFATALS *number*

Maximum Number of Fatal Errors

Compile only. Default: `-maxfatal 50`

Specify `-maxfatal` to limit the number of fatal errors LF95 will generate before aborting.

-ML *target*

Mixed Language

Compile only. Default: `-ml lf95`

Specify the `-ml` switch if your code calls or is called by code written in another language or if your code will call routines in DLLs created by LF95. `-ml` affects name mangling for routine names in `DLL_IMPORT`, `DLL_EXPORT`, and `ML_EXTERNAL` statements. See "Mixed Language Programming" on page 38 for more information.

Syntax

```
-ML target
```

Where:

target is `bc` for Borland C++; `bd` for Borland Delphi; `msvb` for Microsoft Visual Basic; `msvc` for Microsoft Visual C++; `fc` for Fujitsu C; `LF95` for LF95; `LF90` for LF90; and `winapi` for accessing the Windows API directly.

-MLDEFAULT *target*

Mixed Language Default

Compile only. Default: `-mldefault`

Specify the `-mldefault` switch to set the default target language name decoration/calling convention for all program units. `-mldefault` affects name mangling for routine names in `DLL_IMPORT`, `DLL_EXPORT`, and `ML_EXTERNAL` statements.

Syntax

`-MLDEFAULT target`

Where:

target is `bc` for Borland C++; `bd` for Borland Delphi; `msvb` for Microsoft Visual Basic; `msvc` for Microsoft Visual C++; `fc` for Fujitsu C; `LF95` for LF95; `LF90` for LF90; and `winapi` for accessing the Windows API directly.

-MOD *path*

Module Path

Compile only. Default: current directory

Specify `-mod path` to instruct the compiler to search the specified directory for LF95 module files (`.MOD`). New module and module object files will be placed in the first directory specified by *path*. Note that any module *object* files needed from previous compilations must be added to the LF95 command line.

Example

```
LF95 modprog mod.obj othermod.obj -mod ..\mods;..\othermods
```

In this example, the compiler first searches `..\mods` and then searches `..\othermods`. Any module and module object files produced from `modprog.f90` are placed in `..\mods`.

-O0 and **-O1**

Optimization Level

Compile only. Default: `-o1`

Specify `-o0` to perform no optimization. `-o0` is automatically turned on when the `-g` option or the `-chk` option is specified. see “*-[N]G*” on page 23

Specify `-o1` to perform full optimization.

-O *filename*

Object Filename

Compile only. Default: name of the source file with the extension `.OBJ`

Specify `-o name` to override the default object file name. The compiler produces an object file with the specified name. If multiple source file names are specified explicitly or by wildcards, `-o` causes the driver to report a fatal error..

-OUT *filename*

Output Filename

Link only. Default: the name of the first object or source file, with the `.EXE` or `.DLL` extension. The output file is not automatically placed in the current directory. By default it is placed in the same directory as the first object file listed on the command line.

This switch takes a single argument, which is the path and name of the output file. If the file extension `.EXE` is specified, an executable file will be created. If the file extension `.DLL` is specified, a dynamic-link library will be created. If the file extension `.LIB` is specified, a static library will be created. If no extension is specified, `.EXE` is assumed with `-ndl1`; `.DLL` is assumed with `-d11`. If the `-d11` switch is specified, `.DLL` is assumed. If no path information is specified with the file name, then the output file is placed in the current directory.

Example

```
LF95 hello.obj -out d:\LF95\hello.exe
LF95 main.obj -out maintest
```

-/N/PAUSE

Pause After Program Completion

Compile only. Default: `-npause`

Specifying `-pause` will cause the executable program to wait for a keystroke from the user at program completion, before returning to the operating system. This switch can be used to keep the console window from vanishing at program completion, thereby allowing the user to view the final console output. A console window will vanish at program completion if the program is invoked from Windows Explorer or the Start menu, or if the console is generated by a Windows GUI application.

See also

`-WIN` or `-WINCONSOLE`

-/N/PCA

Protect Constant Arguments

Compile only. Default: `-npca`

Specify `-pca` to prevent invoked subprograms from storing into constants.

Example

```
call sub(5)
print *, 5
end
subroutine sub(i)
i = i + 1
end
```

This example would print 5 using `-pca` and 6 using `-npca`.

-/N/PRIVATE

Default Module Accessibility

Compile only. Default: `-nprivate`

Specify `-private` to change the default accessibility of module entities from `PUBLIC` to `PRIVATE` (see “`PUBLIC`” and “`PRIVATE`” statements in the Language Reference).

-PUBList option

Map File Symbol Sort Order

Link only. Default: `-publist byname`

The `-PUBList` switch controls the ordering of the list of public symbols in the map file. The `-PUBList` switch has options to control the ordering of public symbols. They are:

Table 4: -PUBList Options

BYNAME	Sort the list of public symbols which make up the program alphabetically. This is the default operation of the linker.
BYVALUE	Sort the list of public symbols in the program by value. This option is useful when using the map file to find out what routine or variable resides at a particular memory location.
BOTH	Produce two listings of the public symbols: one sorted alphabetically and one sorted by value.
NONE	Cause the linker not to list the public symbols which make up the program at all. This option is useful for reducing the size of the map file.

Syntax

```
-PUBList BYNAME
-PUBList BYVALUE
-PUBList BOTH
-PUBList NONE
```

Example

```
LF95 hello.obj -publ byvalue
```

-/N/QUAD

Quad Precision

Compile only. Default: `-nquad`

Specify `-quad` to extend all double-precision REAL and double-precision COMPLEX variables, arrays, constants, and functions to REAL (KIND=16) and COMPLEX (KIND=16) respectively. If you use `-quad`, all source files (including modules) in a program should be compiled with `-quad`. Specifying `-quad` will usually result in significantly slower executables. All exceptions will be trapped by default. This behavior can be overridden using the NDPEXC routine or the ERRSET service routine (see the file SERVICE.TXT).

-/N/SAV

SAVE Local Variables

Compile only. Default: `-nsav`

Specify `-sav` to allocate local variables in a compiler-generated SAVE area. `-nsav` allocates variables on the stack. `-sav` is equivalent to having a SAVE statement in each subprogram except that `-sav` does not apply to local variables in a recursive function whereas the SAVE statement does. Specifying `-sav` will cause your executable to run more slowly, especially if you have many routines. Specifying `-nsav` may sometimes require more than the default stack (see "*-Stack*" on page 32).

-/N/STATICLINK

Static Link

Compile *and* link. Default: `-nstaticlink`

Specify `-staticlink` with `-win` and `-ml` to link statically with code produced by another supported language system. See "*Mixed Language Programming*" on page 38 for more information.

-Stack

Stack Size

Link only. Default: 100000h bytes (link only)

The `-Stack` switch specifies the size of the stack area for a program. The switch must be followed by a numeric constant that specifies the number of bytes to be allocated to the stack.

If a stack segment is already present in the program, then the `-Stack` switch changes the size of the existing segment. The linker, however, will only increase the size of the existing stack area. If an attempt is made to decrease the size of the stack area, the linker issues an error.

If your program runs out of stack at runtime, increase the stack size with `-Stack`. Stack requirements are noted in the listing file (see "*-/N/LST*" on page 26) Note that some recursive procedures and files with large arrays compiled with `-nsav` can use very large amounts of stack.

Syntax

`-Stack nbytes`

Example

```
LF95 hello.obj -s 200000
```

-/N/STCHK

Stack Overflow Check

Compile only. Default: `-stchk`

Specify `-nstchk` to cause the compiler not to generate code for stack overflow checking. Though your program may execute faster, the stack is not protected from growing too large and corrupting data.

-/N/SWM *msgs*

Suppress Warning Message(s)

Compile only. Default: `-nswm`

To suppress a particular error message, specify its number after `-swm`.

Example

`-swm 16,32`

This example would suppress warning messages 16 and 32. To suppress all warnings, use `-nw`.

-T4, -TP, and -TPP

Target Processor

Compile only. Default: set on installation

Specify `-t4` to generate code optimized for the Intel 80386 or 80486 processor.

Specify `-tp` to generate code optimized for the Intel Pentium or Pentium MMX processors, or their generic counterparts.

Specify `-tpp` to generate code optimized for the Intel Pentium Pro, Pentium II, Pentium III, or Celeron processors, or their generic counterparts. Please note: code generated with `-tpp` is *not* compatible with processors made earlier than the Pentium Pro.

-[N]TRACE

Location and Call Traceback for Runtime Errors

Compile only. Default: `-trace`

The `-trace` switch causes a call traceback with routine names and line numbers to be generated with runtime error messages. With `-ntrace` no line numbers are generated, and the Sampler tool cannot be used (the Sampler tool requires line number information -- see Chapter 8, *The Sampler Tool*).

-[N]TRAP exceptions

Trap NDP Exceptions

Compile only. Default: `-ntrap`

The `-trap` switch specifies how each of four numeric data processor (NDP) exceptions will be handled at execution time of your program.

Table 5: NDP Exceptions

NDP Exception	Switch Argument
Divide-by-Zero	d
Invalid Operation	i
Overflow	o
Underflow	u

Specify `-trap` with any combination of `d`, `i`, `o`, and `u` to instruct the NDP chip to generate an interrupt when it detects the specified exception(s) and generate an error message. Note that trapping cannot be disabled when `-quad` is specified, except by using the NDPEXC routine or the ERRSET service routine (see the file SERVICE.TXT)

Syntax

`-TRAP [d][i][o][u]`

Where:

the `d`, `i`, `o`, and `u` arguments can be used in any combination, as explained above.

-TwoCase and -OneCase

Linker Case Sensitivity

Link only. Default: `-onecase`

For `-OneCase`, the linker ignores the case of public symbols that make up the program being linked. For example, the symbols `abc`, `ABC`, and `aBc` are equivalent in the linker.

The `-TwoCase` switch enables case-sensitive processing of user-defined symbols. When this switch is used, upper- and lower-case versions of the same symbol are considered to be different. `-win` forces `-TwoCase`.

`-OneCase` enforces default behavior.

Example

```
LF95 hello.obj -lib \LF95\graph90 -tc
```

-[N]VSW

Very Simple Windows

Compile *and* link. Default: `-nvsw`

The `-vsw` switch creates a simple console-like Windows GUI application. The window is scrollable.

-[N]W

Warn

Compile only. Default: `-w`

Specify `-w` to generate compile warning and informational messages.

-[NO]WARN and -FULLWARN

Warning Detail

Link only. Default: `-warn`

The linker detects some conditions that can potentially cause run-time problems but are not necessarily errors. Warning messages for these conditions can optionally be generated on the display and in the map file. The linker supports three warning levels: `-WARN`, `-FULLWARN`, and `-NOWARN`.

`-WARN` enables basic linker warning messages.

-FULLWARN enables additional warning messages for the following conditions:

- Multiple initializations of common blocks with different values. The last object module processed is the one that supplies initial values to the output file.
- Pieces of a single segment from different object modules having different segment attributes.
- Inconsistent segment grouping in different object modules.

-NOWARN disables all linker warning messages.

Example

```
LF95 hello.obj -warn
```

-WIN or -WINCONSOLE

Windows

Compile *and* link. Default: `-winconsole`

Specify `-win` or `-winconsole` to create a 32-bit Windows application. Specifying `-winconsole` will create an application for Windows console mode. Viewing of console output may require that `-pause` also be specified in some cases. Under Windows 9x, the `-win` switch can only be used for GUI applications, not console-mode applications (use `-winconsole` for console-mode applications). The `-win` switch requires a WinMain. You can accomplish this by one of the following methods: 1) Use the built-in WinMain in LF95 which will call your LF95 main program; 2) Set up your own WinMain using calls to the Windows API; 3) Build your user interface in another language where the WinMain is set up in that other language (often without the user knowing it, as in Visual Basic and Delphi); 4) Some library packages such as Winteracter and RealWin include a WinMain.

`-winconsole` will put a Windows console (DOS box) on the screen if the program is invoked from Windows explorer or the Start menu. If the program is invoked from the command line of an existing console window, then all console I/O will be performed within that window. In other words, a `-winconsole` program has the “look and feel” of a DOS program running in a DOS box. This is your best choice if you need to see program results after a program run. Note that the console will disappear after program completion if the program is invoked from Windows explorer or the Start menu.

Console output with `-win` is allowed if your program is running on Windows NT. If your program reads from or writes to standard output, a console will be created but the console will immediately disappear after your program runs to completion.

See also

`-/N/PAUSE`

`-/N/WISK`

Winteracter Starter Kit

Compile and link. Default: `-nwisk` (compile and link)

Specify `-wisk` to create an application using the *Winteracter* Starter Kit (WiSK, see the *Winteracter* Starter Kit Manual). Note that a resource file name must be given on the command line whenever specifying `-wisk`. See the *Winteracter* Starter Kit manual for more information.

-/N/WO

Warn Obsolescent

Compile only. Default: `-nwo`

Specify `-wo` to generate warning messages when the compiler encounters obsolescent Fortran 90 features.

-/N/XREF

Cross-Reference Listing

Compile only. Default: `-nxref`

Specify `-xref` to generate cross-reference information in the listing file. By default, cross reference file names consist of the root of the source file name plus the extension `.lst`.

Syntax

`-[n]xref[(spec=sval[,spec=sval]...)]`

Where:

`spec` is `f` for the listing file name, or `i` to include INCLUDE files.

For `f=sval`, the cross reference listing file name, `sval` specifies the cross reference listing file name to use instead of the default. If a file with this name already exists, it is overwritten. If the file can't be overwritten, the compiler aborts.

For `i=sval`, `sval` is one of the characters of the set `[YyNn]`, where `Y` and `y` indicate that include files should be included in the listing and `N` and `n` indicate that they should not. By default, include files are not included in the listing.

Example

```
LF95 myprog -lst -xref(i=y)
```

creates the cross reference file `myprog.lst` and outputs cross reference information for the source file.

See also

`-/N/LST`

-/N/ZERO

Initialize Variables to Zero

Compile only. Default: `-zero`

Specifying `-zero` will cause all variables and data areas to be initialized to zero at program load time, if they are not already initialized by your Fortran code.

Linking Rules

During this phase, the driver submits object files, object file libraries, and compiled resource files (Windows only) to the linker for creation of the executable or dynamic link library.

Fortran 90 Modules

If your program uses Fortran 90 modules that have already been compiled, you must add the module object filenames to the LF95 command line.

Searching Rules

The linker reads individual object files and object module libraries, resolves references to external symbols, and writes out a single executable file or dynamic link library. The linker can also create a map file containing information about the segments and public symbols in the program.

If an object module or library was specified on the command line and contains path information, then it must reside at the location specified. If the path was not specified, the linker looks for the files in the following order:

1. in the current working directory
2. in any directories specified by the `-LIBPath` switch included in the `386LINK` environment variable.
3. in any directories specified with the `-LIBPath` switch (note that `-LIBPath` searches for library files only, not object modules).

Object File Processing Rules

Object modules specified as individual object files are processed in the order in which they appear on the command line.

Library Searching Rules

The order in which object modules from libraries are processed is not always obvious. The linker applies the following rules when searching object libraries:

1. Any libraries specified using the `-Lib` switch are searched in the order in which they appear in the LF95 command string before the LF95 runtime library. The compiler writes the LF95 default library names into each object file it generates.
2. Each library is searched until all possible external references, including backward references within the library, are resolved.
3. If necessary, the linker recursively scans the list of libraries until all external references are resolved.

This algorithm is particularly important when two different object modules in two different libraries each have a public symbol with the same name. If both object modules are linked, the linker signals a duplicate symbol error because they both have public symbols which are referenced elsewhere in the program. However, if the only symbol referenced in both object modules is the duplicate symbol, then only the first object module encountered is linked and no error message is generated. In this latter case, the object module which actually gets linked is determined by applying the rules listed above.

Mixed Language Programming

LF95 code can call and be called by code written in certain other languages. This can be done via dynamic linking (DLLs) or static linking.

Using DLLs

A Dynamic Link Library (DLL) is a collection of procedures packaged together as an executable file, not a library file. Even though it is in the form of an executable, a DLL cannot run on its own. The functions and subroutines in a DLL are called from a `.EXE` file that contains a main program. Note that issuing a `STOP` statement from within a Fortran DLL will cause the entire program to terminate.

With LF95 you can create 32-bit DLLs for use with the language systems in the table below. Console I/O in the Fortran code is not recommended in Windows GUI applications, but just about everything else that is supported under Windows will work. Calls can be made from Fortran to Fortran, from Fortran to another language, and from another language to Fortran. If you are calling DLL routines from a language system other than LF95, please refer to that language system's DLL documentation for more information.

What Is Supported

Lahey/Fujitsu Fortran 95 supports DLL interfaces to the following languages and operating systems (this list is subject to change—see `READ_DLL.TXT` for any changes):

Table 6: Compiler Support for Lahey DLLs

Language System	Version
Lahey/Fujitsu LF95	5.0
Lahey LF90	2.01 and later
Borland C++	4.5 and later
Borland Delphi	2.0 and later
Microsoft Visual C++	2.0 and later
Microsoft Visual Basic	4.0 and later

Your assembly routines may be called from the Fortran routines, however the use of interrupt 21h is not supported. Refer to `README.ASM` for more information regarding interfacing LF95 with assembly code. LF95 can build DLLs callable from Microsoft Visual Basic. Microsoft Visual Basic does not build DLLs callable by LF95.

Declaring Your Procedures

In order to reference a procedure across a DLL interface, the LF95 compiler must be informed of the procedure name and told how to ‘decorate’ the external names in your DLL. The procedure names are defined with the `DLL_EXPORT` and `DLL_IMPORT` statements (see “*DLL_EXPORT Statement*” and “*DLL_IMPORT Statement*” in the LF95 Language Reference). Please note that in general, DLL procedure names are *case sensitive* (unlike the Fortran naming convention, which ignores case). `DLL_EXPORT` is used when defining a DLL and `DLL_IMPORT` is used when referencing a DLL. The type of DLL interface is defined with the

use of the `-ML` compiler switch. You cannot mix `-ML` options in a single invocation of LF95. If you need to reference DLLs from multiple languages you can do so by putting the references in separate source files and compiling them separately. The `-ML` switch options are:

Table 7: -ML Switch Options

Switch	Compiler
<code>-ML LF95</code>	Lahey/Fujitsu Fortran 95
<code>-ML LF90</code>	Lahey Fortran 90
<code>-ML MSVC</code>	Microsoft Visual C++
<code>-ML MSVB</code>	Microsoft Visual Basic
<code>-ML BC</code>	Borland C++
<code>-ML BD</code>	Borland Delphi
<code>-ML WINAPI</code>	Windows API functions invoked directly from Fortran

Building Fortran DLLs

When you create a Fortran DLL, you must indicate the procedures that you want to have available in the DLL with the `DLL_EXPORT` statement. The procedures may be subroutines or functions. When mixing languages, the function results must be of type default `INTEGER`, `REAL`, or `LOGICAL`. The case of the name as it appears in the `DLL_EXPORT` and `DLL_IMPORT` statements is preserved for external resolution except when the `-ML LF90` option is used; within the Fortran code the case is ignored, i.e., `F00` is the same as `FOO`.

To export a procedure from a Fortran DLL, use the `DLL_EXPORT` statement, for example:

```
integer function half(x)
  dll_export half !name is case-sensitive.
  integer :: x
  half = x/2
end
```

This code must be compiled using LF95's `-ml target` switch in order to be callable by language *target* (see "`-ML target`" on page 28).

Note that `DLL_EXPORT` and `DLL_IMPORT` are statements and not attributes. In other words, `DLL_EXPORT` may not appear in an attribute list in an `INTEGER`, `REAL`, `COMPLEX`, `LOGICAL`, `CHARACTER` or `TYPE` statement.

Calling DLLs from Fortran

When you create a Fortran routine that references a procedure in a DLL you declare the DLL procedure name with the `DLL_IMPORT` statement in your Fortran code. The syntax of the `DLL_IMPORT` statement is:

```
DLL_IMPORT dll-import-name-list
```

Where *dll-import-name-list* is a comma-separated list of names of DLL procedures referenced in this scoping unit. The procedures may be subroutines or functions. Non-Fortran DLL routines may only return default INTEGER, REAL, or LOGICAL results.

Use the `DLL_IMPORT` statement as follows:

```
program main
  implicit none
  real :: My_Dll_Routine, x
  dll_import My_Dll_Routine !name is case-sensitive.
  x = My_Dll_Routine()
  write (*,*) x
end program main
```

For further examples, refer to the directories below LF95's EXAMPLES directory.

Passing Data

The only ways to pass data to or from a DLL are as arguments, function results, or in files. LF95 does not support the sharing of data (as with a COMMON block) across the boundaries of a DLL.

Delivering Applications with LF95 DLLs

When you deliver applications built with LF95 DLLs, you must deliver the DLLs you created and any required by the GUI front-end generating tool. All of the DLLs must be available on the path or in a directory that Windows checks for DLLs.

Fortran Calling Fortran DLLs

To create a DLL that works with a Fortran main program, indicate the exported procedure with the `DLL_EXPORT` statement, then run LF95 like this:

```
LF95 source.f90 -win -dll -ml LF95
```

The LF95 compiler builds the DLL `source.dll`. It also builds a `source.imp` file containing the linker commands needed to link to this DLL. Note that the compiler allows you to build your DLL from multiple `.OBJ` files. Remember that the `-DLL` switch is needed on any file that contains a `DLL_EXPORT` statement even if compiled with the `-C` option.

Next build the Fortran Main with:

```
LF95 main.f90 -win -ml LF95 source.imp
```

Ensure that the DLL is available on your path.

C Calling Fortran DLLs

To use Fortran DLLs with Microsoft Visual C++, indicate in the Fortran source the procedures that you want to make available with the `DLL_EXPORT` statements. Remember that the source for the DLL must not have a main program. Then run the LF95 compiler as follows:

```
LF95 source.f90 -win -ml msvc -dll
```

To compile your Fortran source for use with Borland C++, type this:

```
LF95 source.f90 -win -ml bc -dll
```

When LF95 creates a DLL to be called by C, it also creates an import library. Import libraries tell a linker what is available from a DLL. LF95 uses the Microsoft program `LIB` to build an import library for Visual C++. It uses Borland's `IMPLIB` to build the import library for Borland C++. Once you've created the DLL, just link the associated import library (`source.lib` in the above cases) with your C object code, and be sure the DLL is available on your system path.

Fortran Calling C DLLs

Before running the LF95 compiler, you must first build your DLL. Refer to your C manual for specifics. The C compiler builds a `.LIB` file for the DLL.

To compile your Fortran source that calls a Microsoft Visual C++ DLL, type:

```
LF95 source.f90 -win -ml msvc -lib dll_src.lib
```

To compile your Fortran source that calls a Borland C++ DLL, type:

```
LF95 source.f90 -win -ml bc -lib dll_src.lib
```

Where `dll_src.lib` is the name of the import library. Passing arguments from Fortran to a C DLL is done in the same way as for calling the Windows API. For more information, see "*Calling the Windows API*" on page 48.

Referencing DLL Procedures

Fortran functions are called from C as functions returning a value.

For example, this Fortran function:

```

integer function foo(i,j)
integer :: i, j
      :
      :
end function foo

```

uses this C prototype:

```
long foo(long int *i, long int *j);
```

To reference the above function from your C code, declare it with `__stdcall`:

```
long __stdcall foo(long int *i, long int *j);
```

In C++, use:

```
extern "C" {long __stdcall foo(long int *i, long int *j); };
```

For a short example, see `mkvcf90.bat` in LF95's `MIX_LANG\MSVC` directory (for Microsoft Visual C++) or `mkbcf90.bat` (for Borland C++) in LF95's `MIX_LANG\BC` directory.

Passing Arguments from C or C++

Subroutines and default INTEGER, REAL, and LOGICAL function types are supported.

Lahey's calling conventions are as follows:

- All arguments are pass-by-address, not pass-by-value as in C.
- Arrays of pointers cannot be passed.
- COMPLEX and derived type arguments can be passed as pointers to structures. For COMPLEX, these structures are:

```

typedef struct {
    float real;
    float imaginary;
} complex;

```

```

typedef struct {
    double real;
    double imaginary;
} double_complex;

```

- Character arguments are passed as pointers to strings. When a Fortran program unit contains character dummy arguments with `len=*`, then any routine calling that program unit must append to the end of the argument list the length of each of the corresponding actual arguments. The lengths must be passed by value.

For example, the Fortran subroutine:

```
subroutine example3 (int1, char1, int2, char2, char1_len)
  integer int1, int2, char1_len
  character (len=char1_len) :: char1
  character (len=25) :: char2
end
```

would have this prototype in C:

```
void example3 (long int *int1, \
              char *char1, \
              long int *int2, \
              char *char2, \
              long int char1_len);
```

Passing Arrays in C or C++

Because C processes arrays as an array of arrays and Fortran processes arrays as multi-dimensional arrays, there are some special considerations in processing a Fortran array. Excluding a single-dimension array (which is stored the same in C as in Fortran), you will need to reverse the indices when accessing a Fortran array in C. The reason for this is that in C, the right-most index varies most quickly and in Fortran the left-most index varies most quickly (multi-dimensional). In an array of arrays, the columns are stored sequentially: row 1-column 1 is followed by row 1-column 2, etc. In a multi-dimensional array, the rows are stored sequentially: row 1-column 1 is followed by row 2-column 1, etc.

Also note that all C arrays start at 0. We do not recommend that you use a lower dimension bound other than zero (0) as your C code will have to modify the indices based on the value used. We strongly recommend that you do not use negative lower and upper dimension bounds!

If the subscript ranges are not known at compile time, they can be passed at runtime, but you will have to provide the code to scale the indices to access the proper members of the array.

Some sample code may help explain the array differences. Your Fortran code would look like:

```

subroutine test(real_array)
real :: real_array(0:4,0:5,0:6,0:7,0:8,0:9,0:10)
integer :: i,j,k,l,m,n,o
do o = 0, 10
  do n = 0, 9
    do m = 0, 8
      do l = 0, 7
        do k = 0, 6
          do j = 0, 5
            do i = 0, 4
              real_array(i,j,k,l,m,n,o) = 12.00
            end do
          end do
        end do
      end do
    end do
  end do
end do
end subroutine test

```

The equivalent C code would look like:

```

void test(float real_array[10][9][8][7][6][5][4])
int i,j,k,l,m,n,o;
/*
** this is what the subscripts would look like on the C side
*/
for(o = 0; o < 11; o++)
  for(n = 0; n < 10; n++)
    for(m = 0; m < 9; m++)
      for(l = 0; l < 8; l++)
        for(k = 0; k < 7; k++)
          for(j = 0; j < 6; j++)
            for(i = 0; i < 5; i++)
              real_array[o][n][m][l][k][j][i] = 12.000;
  return;
}

```

On the Fortran side of the call, the array argument must not be dimensioned as an assumed-shape array. You should use explicit shape, assumed size, or adjustable arrays.

Microsoft Visual Basic Information

To create a DLL that will work with Microsoft Visual Basic, take Fortran source (without a main program) and indicate the procedures that you want available in the DLL with the `DLL_EXPORT` statement, then invoke the LF95 driver like this:

```
LF95 source.f90 -win -dll -ml msvb
```

Declaring your Procedure in Visual Basic

In your BASIC code, a procedure's declaration will be like one of the following examples:

```
Private Declare Function my_func Lib "my_dll" (ByRef my_arg As Long) As Long
Private Declare Sub my_sub Lib "my_dll" (ByRef my_arg As Long)
```

(see the relevant section below if an item on the argument list is either an array or is character datatype). Note that in the example above, "my_dll" must specify a complete path in order to operate within the Visual Basic Environment.

Passing Character Data in Visual Basic

Character arguments are passed as strings with the length of each string appended at the end of the argument list.

Character (string) arguments and hidden length arguments must be passed by value, i.e., declare the procedure's arguments (actual and hidden) with the `ByVal` keyword. Refer to the example `VBDEMO` program. The following restrictions apply:

- Character arguments should be declared as `CHARACTER(LEN=*)`.
- Fortran functions returning character data to Visual Basic are not supported.

Passing Arrays in Visual Basic

When passing an array from Microsoft Visual Basic you will need to declare the argument as a scalar value in the Basic declaration, and pass the first element of the array as the actual argument. Declare the array dummy argument normally in the Fortran procedure. Note that the default lower bound for arrays in Visual Basic is 0, so you may find it helpful to explicitly declare your Fortran arrays with a lower bound of 0 for each dimension, or explicitly declare your Basic arrays to have a lower bound of 1 (this can be done at the module or procedure level via the `Option Base` statement). Note also that arrays of strings cannot be passed from Visual Basic to LF95.

Running the Visual Basic Demo

1. Compile the `VBDEMO.F90` file, located in LF95's `MIX_LANG\MSVB` directory, using the `-dll -win -ml msvb` switches.
2. Ensure that the resulting `VBDEMO.DLL` resides in a directory that is on your path. Failure to do this will generally result in an "Error loading DLL" message from the operating system.
3. Start Visual Basic and open the `VBDEMO.VBP` project in LF95's `MIX_LANG\MSVB` directory.
4. Run the demo (F5).

Borland Delphi Information

Passing Character Data in Delphi

Character arguments are passed as strings with the length of each string appended at the end of the argument list.

Delphi has two kinds of strings: long strings and short strings, where a long string can contain a very large number of characters and its length varies dynamically as needed, and a short string has a specified length and may contain up to 255 characters. If your character argument is a short string you should use the `var` keyword in your procedure's declaration; omit the `var` keyword if your argument is a long string. Refer to the `BDDEMO` and `BDDEMO2` programs to see examples for both of these cases.

As of this writing, the following conditions apply:

- Character arguments should be declared as `CHARACTER(LEN=*)`.
- “Long string” character arguments should be treated as `INTENT(IN)`.
- “Short string” character arguments may be treated as `INTENT(IN OUT)`.
- Fortran functions returning `CHARACTER` data to Delphi are not supported.

Passing Arrays in Delphi

Because Delphi processes multi-dimensional arrays as an array of arrays (like C and C++) and Fortran processes arrays as multi-dimensional arrays, there are some special considerations in processing a Fortran array. Refer to the “Passing Arrays in C or C++” section for more information.

Delphi Calling Fortran

To create a DLL that will work with Borland Delphi, take the Fortran source (without a main program) and indicate the procedures that you want available in the DLL with the `DLL_EXPORT` statement, then invoke the `LF95` driver like this:

```
LF95 source.f90 -win -dll -ml bd
```

Declaring your Procedure in Delphi

In your Delphi code, a procedure's declaration will be like one of the following examples:

```
function my_LF95_function(var my_arg: LongInt) : LongInt;
  stdcall; external 'my_dll.dll';
procedure my_LF95_subroutine( var my_arg: Single); stdcall;
  external 'my_dll.dll';
```

(see the relevant section below if an item on the argument list is either an array or is character datatype).

Running the Delphi Calling Fortran Demo

1. Compile the `BDDEMO2.F90` file located in `LF95's MIX_LANG\BD` directory using the `-dll`, `-win`, and `-ml bd` switches.

2. Ensure that the resulting `BDDEMO2.DLL` resides in a directory that is on your path. Failure to do this will generally result in an “Debugger Kernel Error” message from the operating system.
3. Start Delphi and open the `BDDEMO2.DPR` project in LF95’s `MIX_LANG\BD` directory.
4. Run the demo (F9).

Fortran Calling Delphi DLLs

Before running the LF95 compiler, you must first build your DLL. Refer to your Delphi manual for the specifics. Because Delphi does not build a `.LIB` file for the DLL, you will need to specify the imported names on the command line.

To compile a Fortran routine to call a Delphi DLL:

```
LF95 main.f90 -win -ml bd -implib my_dll.dll -import func1  
      func2 ...
```

where `main.f90` is the fortran program which calls `func1` and `func2` in `my_dll.dll`.

Running the Fortran Calling Delphi Demo

1. From Delphi, open `F90CALBD.DPR` in LF95’s `MIX_LANG\BD` directory.
2. Build the DLL by pressing `Ctrl-F9`.
3. Copy `F90CALBD.DLL` to LF95’s `MIX_LANG\BD` directory.
4. Change to LF95’s `MIX_LANG\BD` directory.
5. Run the compiler as follows:

```
LF95 f90calbd.f90 -win -ml bd -implib f90calbd.dll -import  
      bd_min bd_max
```

6. Run the resulting executable, `F90CALBD.EXE`

Examples

Please refer to the examples in the directories below LF95’s `EXAMPLES` directory for further information on using the Fortran DLL interface.

Calling the Windows API

See the file `READ_API.TXT` for information on making direct calls to the Windows API.

Static Linking

Linking statically gives a single executable file that contains all of the executable code and static data in the program. LF95 can link statically with code produced with Microsoft Visual C/C++ or Borland C/C++. Information on static linking is the same as for dynamic linking (described above) with the following exceptions:

1. First make sure your LIB environment variable points to your C library directory.
2. Specify the `-staticlink` and `-ml` switches on the LF95 command line (do not specify `-dll`).
3. Use `ML_EXTERNAL` instead of `DLL_IMPORT` or `DLL_EXPORT` in your Fortran source.
4. You must have a Fortran main program.
5. Unlike with DLLs, Fortran common blocks can be accessed from within a statically linked C routine. See the file `FTOC.BAT` for more information.
6. Import libraries and `.imp` files do not need to be included on the LF95 command line (import libraries and `.imp` files are specific to DLLs).
7. Fortran common blocks can be accessed from C when the C is statically linked (this is not possible with a DLL). If you have a common block called `common_name` or `COMMON_NAME`, access it in C as a structure variable called `common_name_` (note the trailing underscore). For example, reference:

```
common /my_common/ a, b, c
real a, b, c
```

as:

```
extern struct
{
float a, b, c;
} my_common_; /* my_common must be all lower case */
```

Fortran common blocks are aligned on one-byte boundaries. To align your C structures along one-byte boundaries, use the `/Zp1` switch or the `pack` pragma with Microsoft Visual C++. Use the `-a-` switch or the `option -a-` pragma with Borland C++. Note that use of these switches should be limited to files or sections of code that require one-byte alignment; one-byte alignment can cause slower access to C structure members.

For more information, see the examples in LF95's `EXAMPLES\MIX_LANG\MSVC` and `EXAMPLES\MIX_LANG\BC` directories.

OpenGL Graphics Programs

OpenGL is a software interface for applications to generate interactive 2D and 3D computer graphics independent of operating system and hardware operations. It is essentially a 2D/3D graphics library which was originally developed by Silicon Graphics with the goal of creating an efficient, platform-independent interface for graphical applications (Note: OpenGL is a trademark of Silicon Graphics Inc.). It is available on many Win32 and Unix systems, and is strong on 3D visualization and animation.

f90gl is a public domain implementation of the official Fortran 90 bindings for OpenGL, consisting of a set of libraries and modules that define the function interfaces. The library, module files, demonstration programs, and documentation are documented in the file `readf9gl.txt`. The f90gl interface was developed by William F. Mitchell of the Mathematical and Computational Sciences Division, National Institute of Standards and Technology, Gaithersburg, in the USA. For information on f90gl, see the F90GL web page at <http://math.nist.gov/f90gl>.

Until recently, the OpenGL LF9x applications could only be built as statically linked applications targeted for Visual C. A much friendlier method is now available thanks to a porting effort implemented by Lawson B. Wakefield of Interactive Software Services Ltd. in the UK. (ISS are the developers of the INTERACTER & Winteracter GUI/graphics Fortran development tools). This implementation has made the OpenGL interface available within the framework of the WISK and Winteracter libraries. A full set of examples is available under the WISK directory of the LF95 installation.

Recommended Switch Settings

Inspect the LF95.FIG file to determine current switch settings.

For debugging, we recommend the following switch settings:

```
-chk (a,e,s,u,x) -chkglobal -g -pca -stchk -trace -w -info
```

(Note: Specifying `-chkglobal` or `-chk (x)` must be used for compilation of all files of the program, or incorrect results may occur.)

For further analysis during development, and consider specifying any of following switch settings:

```
-ap -co -cover -f95 -info -lst -wo -xref
```

For production code, we recommend the following switch settings:

```
-nap -nchk -nchkglobal -ncover -ng -ol -npca -nsav -nstchk  
-ntrace
```

Use `-t4`, `-tp`, or `-tpp` depending on your preferred target processor.

3

Editing and Debugging with ED

Lahey ED for Windows (ED) is a Fortran-smart development environment. You can edit, compile, link, and run your programs all from within ED. ED offers sophisticated editing features like syntax highlighting, intelligent procedure browsing, code completion, macros, drag and drop, as well as the standard editing features you are accustomed to. After a compilation, ED highlights the exact location of programming errors diagnosed by the compiler.

You can also debug Windows programs with ED. While debugging, you can watch the values of variables change during program execution and set breakpoints with a mouse click.

This chapter assumes a basic familiarity with Windows. It presents an overview of ED's functionality. For detailed information, please see ED's on-line help.

Setting Up and Starting ED

Startup

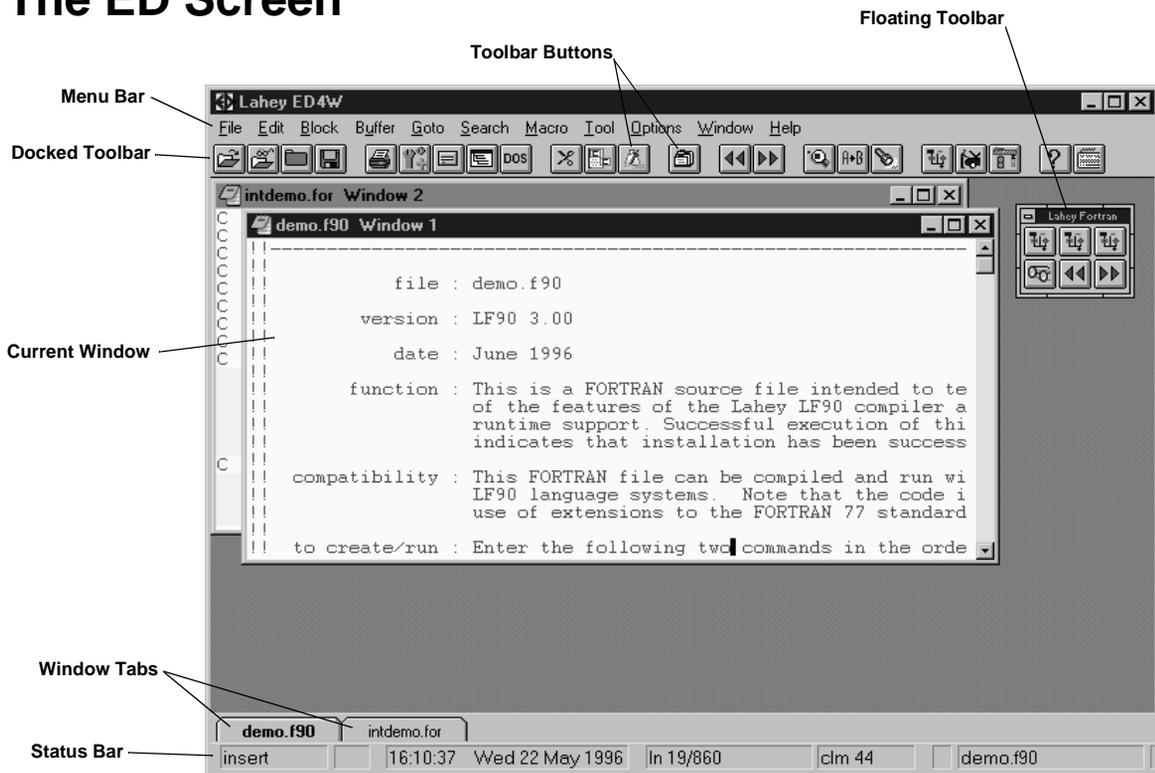
ED must be run from Windows. Start ED by double-clicking the Lahey ED for Windows icon.

Exiting ED

To exit from ED, choose **File | Exit** from the menu, double-click the system menu icon in the top left corner, or press **Alt-F4**.

Always exit ED before turning off your computer. ED checks for unsaved changes and enables you to save these before exiting.

The ED Screen



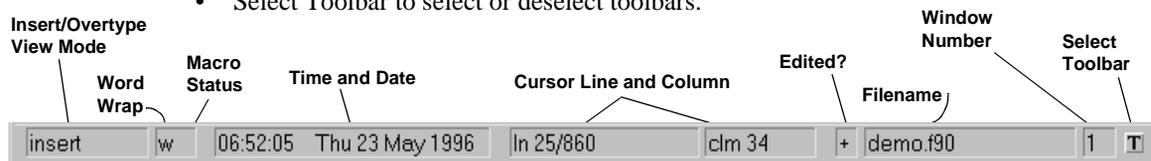
The Menu Bar

ED features pull-down menus from which the various ED commands can be invoked. To open a menu, click on item on the menu bar with the mouse or press **Alt-*underlined letter***. Select a command by clicking on it with the mouse or by pressing *underlined letter*.

The Status Bar

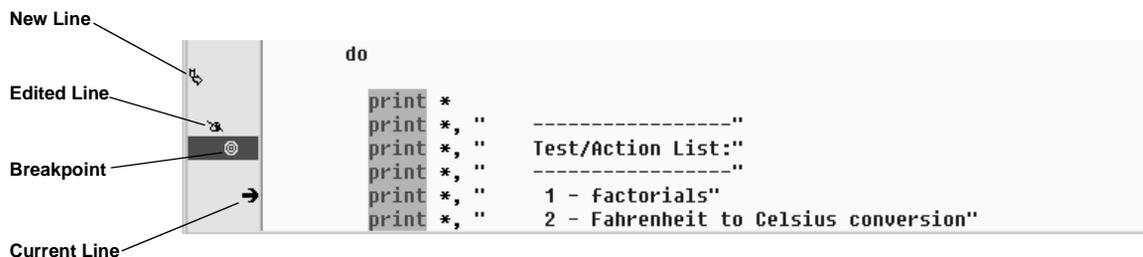
The status bar displays information about the current state of ED. It is more than informational. Click on:

- Insert/Overtyping/View to toggle Insert/Overtyping/View mode.
- the Word Wrap Status to toggle word wrap on or off.
- the Macro Status to play the current macro.
- the Time and Date to change the time and date format.
- the Cursor Line and Column to go to a particular line and column.
- Edited to save the current file.
- the Filename to change the active window.
- Select Toolbar to select or deselect toolbars.



The Text Bar

ED's Text Bar provides a visual reminder of lines that have been edited and lines that have been added. When debugging (see "*Debugging*" on page 61), the Text Bar marks the current line and lines with breakpoints. You can activate the Text Bar by right-clicking on white space to the left of any line, or by selecting Options|Configuration|Display and checking the Text Bar Visible box.



Toolbars

Toolbars provide a quick way to issue commands with the mouse. ED comes with a variety of toolbars. Display different toolbars by pressing the Select Toolbar button on the status bar.

Point to a toolbar button to pop up quick help on that button. To issue the command, left-click.

The Window Bar

ED's window bar makes switching between open files easy. Left-click on a window tab to make that window current. Right click on a tab to perform other operations on that file.

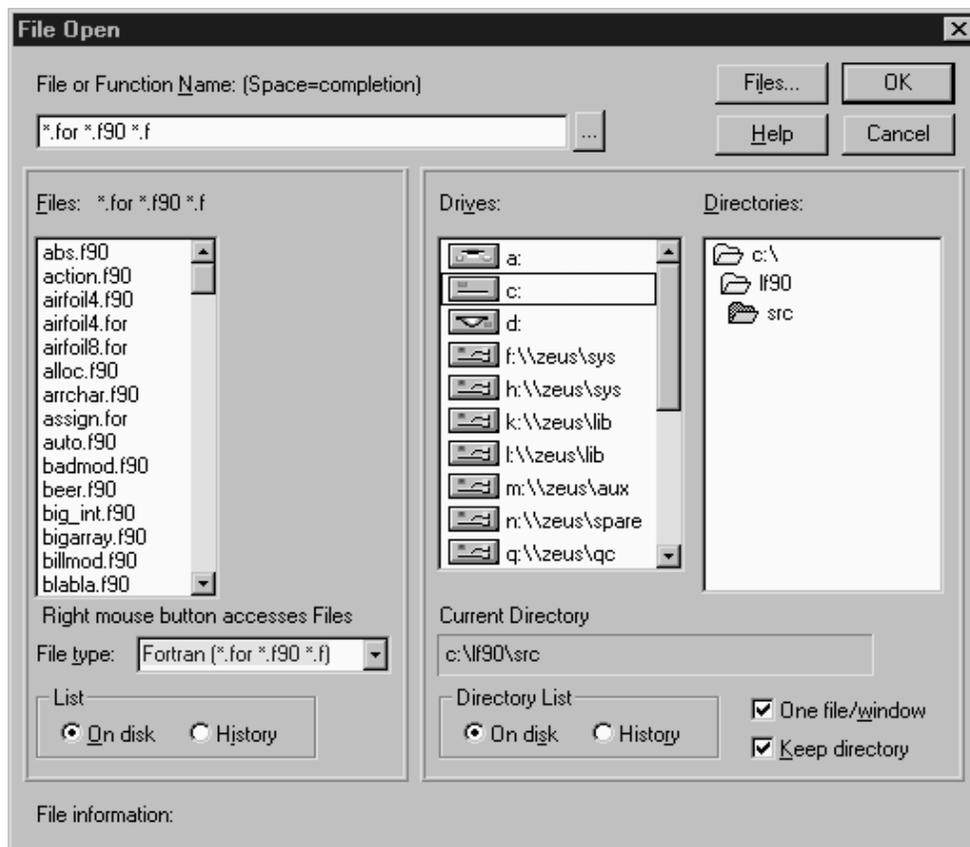
Getting Help

Get general help on ED by pressing `F1`. Get context-sensitive help by pressing `Shift-F1`. Both general and context-sensitive help are also available through the Help menu. A quick tour of ED is available through `Help|Help Contents`. This is especially valuable if you are new to ED and would like to get a glimpse of the features it offers. You can also view a quick reference to ED's shortcut keys by pressing `Alt-Q` or by selecting `Help|Quick Reference`.

Managing Files

Creating A File From Scratch

To create a new file, select `File|New`. Select a file type (file extension). You now have an empty file you can edit. Save the file by selecting `File|Save` or by clicking on the `Save file` button on the docked toolbar. You can also create a new file by clicking on the `Open File for Editing` button on the docked toolbar. The `File|Open` dialog box appears. Select a drive and directory and enter a name for the file. Click `OK` to create the file.



Opening A File

Opening a file that already exists is also done using `File|Open`. You can choose to select directories and files either from a list of those that exist or from a history of those most recently used.

Files are opened in edit mode by default. To open a file in view-only mode, select `File|View`. Toggle between edit and view modes by pressing `Alt-F3`.

If you use include files, you can view them by right clicking on the filename in the `INCLUDE` line in your code.

Syntax Highlighting

ED highlights Fortran syntax elements, like keywords, literal constants, and comments, with different colors. This facility can be customized in several ways. To change the default colors, select `Options|Color Setup`. To change the elements that ED highlights or to add new elements of your own, edit the file `lang_for.iii`, where `iii` are your initials.

By default, only files that end in `.for`, `.f90`, and `.f` use Fortran syntax highlighting. Other file extensions are set to use no syntax highlighting. You can turn on syntax highlighting for some 30 other languages by selecting `Options|File Extension Setup`. Only files setup with Fortran syntax highlighting or with no syntax highlighting can be saved, however.

Navigation

After you have opened a file, move through it using:

- the normal keyboard navigation keys (`PageUp`, `PageDown`, `Home`, `End`, etc.)
- the mouse on the scroll bar
- `Goto|...` to jump to a particular kind of location. The `Goto` menu lists many locations to jump to, most with quick keyboard equivalents.

Previous/Next Procedure

You can quickly move through your file by jumping from procedure to procedure using the `Ctrl-PageUp` and `Ctrl-PageDown` keys.

Function/Procedure List

ED will create a list of procedures in your file if you:

- Choose `Goto|Function List;`
- Press `Alt-F1;` or
- Click the Function/procedure list button on the toolbar

You can then jump directly to any procedure by double clicking on its name in the Function/procedure list.

Find

The Search menu provides a variety of ways to find and optionally replace text. You can search for exact text or use regular expressions, where wildcard characters (* and ?) can be inserted in your search string. You can also search incrementally. `Incremental search` finds the next occurrence of text as you type it.

Matching Parentheses and Statements

You can move automatically to a matching object (a parenthesis or statement) by right clicking on an object. For example, in an expression, right-clicking on a parenthesis will jump to the matching parenthesis. This is a real time-saver in putting together complicated error-free expressions. Right-clicking on a DO statement jumps to the corresponding END DO statement. Right-clicking on an IF statement jumps in succession to any ELSE or ELSE IF statements and ultimately to the corresponding END IF statement.

Editing

To toggle between edit and view modes use the `Alt-F3` key. When in view mode the file is protected from changes and is not locked, permitting other people to view the file at the same time.

Undo and Redo

You can undo any editing or cursor movement in ED. To undo the previous operation, select `Edit|Undo` or press `Ctrl-Z`. To redo the last operation you have undone, select `Edit|Redo` or press `Ctrl-Y`.

Extended Characters

To type characters outside the range of your keyboard:

- Be sure the NumLock key is on.
- Type `Alt-0ANSI character code` to enter Microsoft Windows ANSI font characters above 127.
- Type `Alt-OEM character code` to enter DOS extended characters above 127.
- To enter characters below 32 such as `^A`, `^B`, `^C`, etc., select `Edit|Verbatim Character`, then type the character. This prevents the control key sequence from being interpreted as an ED command.

Blocks

A block is an area of marked text. Once marked, a block can be deleted, moved, copied, searched, sorted, or printed.

Block Operations

ED recognizes three kinds of blocks: stream, line, and column:

- *Stream blocks* are the usual Windows marked blocks. They begin and end at any locations you choose and include all characters in-between.
- *Line blocks* mark whole lines.
- *Column blocks* mark a rectangular area.

Marking a Block with the Keyboard

Position the cursor at the start of the block. Select `Block` and then `Stream Block`, `Line Block`, or `Column Block`. Expand the block using the arrow keys, `PageUp`, and `PageDown`.

Marking a Block with the Mouse

Simply click and drag to mark a block with the mouse. To toggle between the three different kinds of blocks, click the right mouse button while still holding down the left mouse button.

Marking a Word or Line

To mark a word, double-click on the word. Double click in white space left or right of a line to mark the whole line.

Drag and Drop

You can move a marked block to a new location by clicking on the marked block, holding the mouse button down while you move the block, then releasing the mouse button at the desired location. You can copy rather than move a block by holding down the control key while you drag and drop.

Coding Shortcuts

Words or constructs that you type repeatedly can be entered automatically or finished for you by ED.

Templates

Templates are abbreviations for longer sequences of characters. These sequences can be one or a few words or can comprise several lines. Choose `Options|Language Words & Templates` then press `Ctrl-PageDown` to view the Fortran templates. When ED is installed a file called `lang_for.iii` (where `iii` are your initials) is set up. To modify existing templates or add new ones, edit this file.

Templates are expanded by pressing `Esc` or `Space` after typing a template abbreviation. You can specify the number of characters before `Esc` or `Space` will cause template expansion by selecting `Options|File Extension Setup|Templates & Word Wrap`.

Smartype

Smartype recognizes words you have already typed in your file. If you type the first few characters of a word that appears earlier in the file, then press `Esc` or `Space`, the word will be automatically completed for you. If there is an ambiguity you will be presented with a menu of words to select from.

Smartype can be deactivated for the `Space` key by turning off `Space does Smartype` in `Options|File Extension Setup|Templates & Word Wrap`.

Case Conversion

Case conversion changes the case of words you type to match instances of words earlier in the file or to match words in the `lang_for.iii` file. In this way ED ensures consistency in capitalization.

Case conversion can be toggled off or on in `Option|File Extension Setup`.

Code Completion

Code completion finishes open nesting levels introduced by keywords or parentheses. Code completion is activated with the `Esc` key. For example, if you've typed:

```
a = b * ( c + ( d - e
```

pressing `Esc` once will give

```
a = b * ( c + ( d - e )
```

and pressing `Esc` again will give

```
a = b * ( c + ( d - e ) )
```

Compiling from ED

Compiling Your Program

There are two ways of compiling your program while in ED. You can click on the Lahey LF95 button from the toolbar, or select `Tools | Programs | Lahey/Fujitsu LF95 | Run`. When you do so, a window will appear that captures the compiler's output and shows the compiler's progress through your source file. Pressing the F4 key will tile the source and compile windows.

Alternately, you can select the DOS button and compile as described in "*Developing with LF95*" on page 11. Redirecting the compiler's output to the `errs.iii` file (where `iii` are your initials) will enable you automatically to locate errors in your source, as described below.

Locating Errors

ED automatically synchronizes your program source with errors detected by the compiler. To browse through errors, choose `Goto | Next Error` or `Goto | Previous Error`, click on the next or previous error button on the toolbar, or press `Alt-UpArrow` or `Alt-DownArrow`. ED automatically moves the cursor to the appropriate location in your source file.

Changing Compiler Options

To change the LF95 compile line used by ED, select Tool|Programs, then select the 'General' radio button, then select Lahey/Fujitsu LF95|Edit. The following variables may be used in the Command Line and Working Directory fields.

Table 8: Command Line Variables

Variable	Evaluates to
<NAME>	Name of the current file including drive and path.
<PATH>	Drive and path for the current file.
<FILE>	Filename of current file without drive\path and file extension.
<EXT>	Filename extension.
<CWD>	Current working directory.
<ED_DIR>	ED executable's directory.
<ENTER>	Prompts the user for a filename.
<1>	User's response from first <ENTER> .
<2>	User's response from second <ENTER> .
<3>	User's response from third <ENTER> .
<DATE>	Current system date.
<INITIALS>	Your initials.
<WORD>	The word at the current cursor position.

Debugging

Lahey ED for Windows' integrated debugger can run your program, set breakpoints, step a line at a time, view the values of variables in your program in several different ways, and change the values of variables while running your program. The current executable line and any breakpoints are indicated with markers in the Text Bar (see *"The Text Bar"* on page 53).

Starting the Debugger

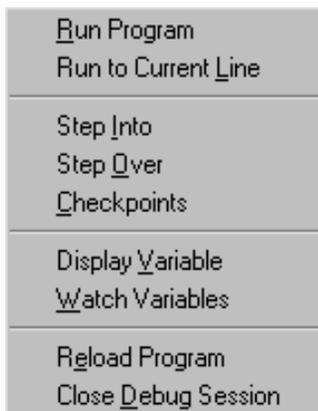
Note that with version 5.5 of LF95, the Lahey/ED debugging facility is off by default and is replaced by WinFDB. However, you can still use the Lahey/ED debugger by selecting the SOLD95 Toolbar from within ED using Options|Toolbars.

Before debugging your program you must compile it using the `-g` and `-winconsole` or `-win` switches (see “*Changing Compiler Options*”, above, and “*Compiler and Linker Switches*” on page 17). The `-g` switch creates an additional file with debugging information. This file ends with the extension `.ydg`. The `-win` or `-winconsole` switch creates a Windows executable file. Start debugging by clicking on the `Debug Program` button in the Lahey Fortran toolbar, or by selecting `Tool|Debug`. Note that Lahey ED can be used to debug both LF95 and LF90 programs. If you will use LF95’s Lahey ED to debug LF90 executables, you must first delete any `.ydg` files that exist for these executables in the same directory.

It is most convenient always to have the Lahey Fortran toolbar visible while debugging. When prompted, enter the name of the executable file, including the filename extension (`.exe`) and, if the file is not in ED’s current directory, a path. For example, if the executable file `myprog.exe` is in a directory called “`programs`” below the root, you would enter

```
\programs\myprog.exe
```

ED will expand the Text Bar and put the current line icon next to the first executable line in your program. It will also open the SOLD95W Output Window so that you can view actions performed by the debugger. Once your program is loaded, click on the `Debug Program` button again to bring up the debug menu. You can also bring up the debug menu by selecting `Tool|Debug` or by right-clicking in ED’s Text Bar.



Running Your Program

To run your program, select `Run Program` from the debug menu, press the F6 key, or click on the `Run Program` button. Just running your program is not particularly useful. You will want your program to stop occasionally by setting breakpoints or by running a line at a time. In this way you can inspect or change the values of variables at troublesome locations in your program.

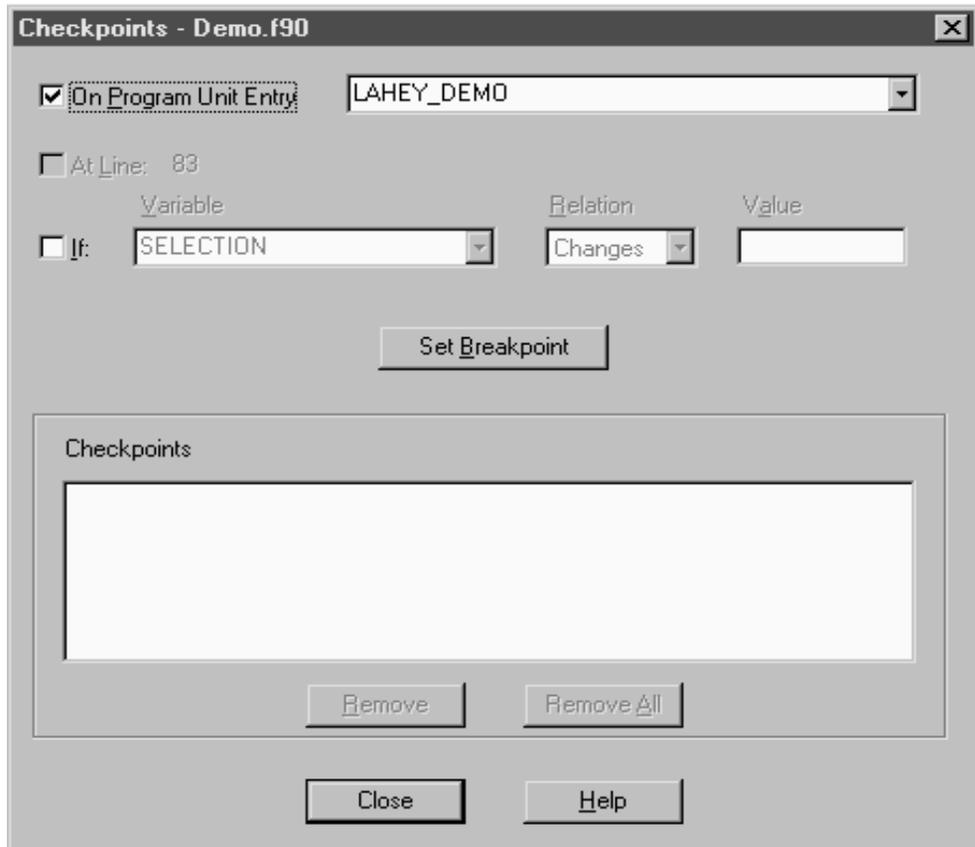
To run to a particular line in your program, click on that line, then select `Run to Current Line` from the debug menu.

Running a Line at a Time

To execute the current executable line in your program (marked with the current line marker in the Text Bar), click on either the `step Into` or `Step Over` button, or select `Step Into` or `Step Over` from the debug menu. If the current line is a subprogram invocation, `Step Into` will execute the current line and move the current line marker to the first executable line in the subprogram; `Step Over` will execute the entire invoked subprogram and move the current line marker to the next executable line in the current subprogram.

Setting Breakpoints

Often when debugging you will want to have your program stop at a particular location in your source code, or when a particular condition occurs. Such a stopping place is called a breakpoint. To set a breakpoint, click on the `Checkpoints` button, or select `Checkpoints` from the debug menu. The following dialog box displays:



To set a breakpoint on entry to a particular program unit, click on the `On Program Unit Entry` checkbox, select the program unit from the list of program units, then click on `Set Breakpoint`. The breakpoint will display in the `Checkpoints` list. You can remove a breakpoint by highlighting it in the list and clicking `Remove`.

To set a breakpoint on a particular line, first click on that line in your source, open the `Checkpoints` dialog and click the `At Line:` checkbox. Then click on `Set Breakpoint`. A shortcut for this procedure is to simply left-click in the `Text Bar` next to the desired line.

To set a breakpoint on a particular condition, click the `If:` checkbox. You can set breakpoints based on whether a condition holds true, such as if `a` is greater than 100, or based on when a value changes.

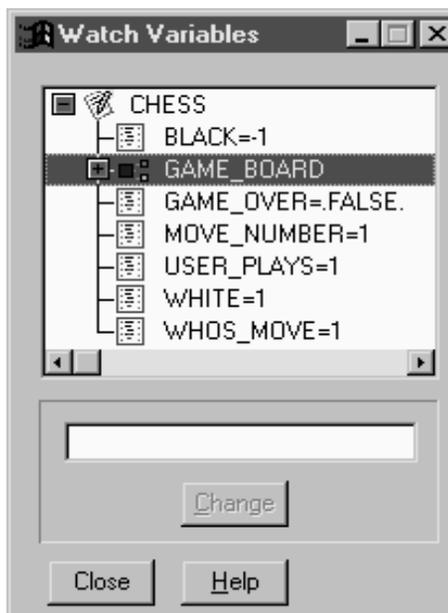
You can remove all breakpoints at once by clicking on the `Remove All` button in the `Checkpoints` dialog.

Displaying the Values of Variables

Values of variables can be displayed three different ways. To display the value of a scalar variable of an intrinsic type (REAL, COMPLEX, INTEGER, LOGICAL, or CHARACTER, as opposed to a derived type), simply right click on the variable name. A box will appear with the name of the variable and its value.

To display the value of any variable, including derived types and arrays and combinations thereof, click on the `Display Variable` button or select `Display Variable` from the debug menu. Select the variable from the list of variables provided. The name and value of the variable appear in the SOLD95W Output Window.

To watch values of variables change as you step through your code, open the Watches Dialog by clicking on the `Open Watches Dialog` button or by selecting `Open Watches Dialog` from the debug menu. A Dialog like the following displays:



In the above Watch Dialog, `GAME_BOARD` is an array. To expand the values of all elements in the array, click on the '+' next to `GAME_BOARD`. Click on '-' to collapse it again. This will work for variables of derived type as well. The Watch Dialog can be resized by dragging its edges or corners with the mouse.

It is recommended that large arrays be kept collapsed while stepping through your program, as updating the values of the elements of large arrays while stepping is time consuming.

Changing the Values of Variables

You can change the value of a variable in a Watch Dialog (see above) by clicking on it, then changing its value in the space below and clicking Change. Only scalar data, single elements of arrays, and single components of derived types can be changed in this way.

Reloading your Program

To reload your program, select `Reload Program` from the debug menu. All checkpoints will remain in effect.

Configuration

To change ED's configuration select `Options | Configuration` or press `F11`. Click on the appropriate tab to display and change a configurable item. Pressing the help button while in one of the configuration dialog boxes brings up context-sensitive help on any of the features in the dialog box.

Status Line

Controls display of the time, date, and window bar.

Display

Controls the display of windows, control characters, menus, and toolbars.

Fonts

Controls the type and size of the display font.

Colors

Controls the colors for syntax highlighting of your source.

Session Info

Controls various parameters for the way ED starts, including whether to restore the current directory, whether to save the desktop state automatically on exit, how large a cache to use, and how large a history list to keep.

Behavior

Controls keyboard emulation, cursor movement and insert/replace behavior.

Safety

Controls how backup copies of your source file are kept.

Locking

Controls how files are locked when accessed on a LAN.

4

Command-Line Debugging with FDB

FDB is a command-line symbolic source-level debugger for Fortran 95, C, and assembly programs. Use FDB if you feel more comfortable with a command-line debugger than with the WinFDB Windows debugger, or if you need access to debugging commands not available in WinFDB.

Before debugging your program you must compile it using the `-g` switch (see "Compiler and Linker Switches" on page 13). The `-g` switch creates an additional file with debugging information -- this file has the same name as the executable with the extension `.ydg`. Debugging cannot be performed without the presence of the `.ydg` file in the same directory as the executable file. FDB cannot be used on LF90 executables.

Starting FDB

To start FDB type:

```
FDB exefile
```

Where: *exefile* is the name of an executable file compiled with the `-g` option.

Commands

Commands can be abbreviated by entering only the underlined letter or letters in the command descriptions. For example, `kill` can be abbreviated simply `k` and `oncebreak` can be abbreviated `ob`. *All commands should be typed in lower case, unless otherwise noted.*

Executing and Terminating a Program

run *arglist*

Passes the *arglist* list of arguments to the program at execution time. When *arglist* is omitted, the program is executed using the arguments last specified. If *arglist* contains an argument that starts with "<" or ">", the program is executed after the I/O is redirected.

Rn

Executes the program without arguments. The “R” should be upper case.

kill

Forces cancellation of the program.

param commandline *arglist*

Assign the program’s command line argument list a new set of values

param commandline

Display the current list of command line arguments

clear commandline

The argument list is deleted

quit

Ends the debugging session.

Shell Commands

cd *dir*

Change working directory to *dir*

pwd

Display the current working directory path

Breakpoints

General Syntax

```
break [location [ ? expr ]]
```

Where *location* corresponds to an address in the program or a line number in a source file, and *expr* corresponds to a conditional expression associated with the breakpoint. The value of *location* may be specified by one of the following items:

- [*'file'*] *line* specifies line number *line* in the source file *file*. If omitted, *file* defaults to the current file.
- *proc* [+|- *offset*] specifies the line number corresponding to the entry point of function or subroutine *proc* plus or minus *offset* lines.
- [*mod*@]*proc*[@*inproc*] specifies function or subroutine *proc* in current scoping unit, or internal procedure *inproc* within *proc*, or procedure *proc* contained in module *mod*.
- **addr* specifies a physical address (default radix is hexadecimal).
- If *location* is omitted, it defaults to the current line of code

The conditional expression *expr* can be constructed of program variables, typedef elements, and constants, along with the following operators:

Minus unary operator (-)

Plus unary operator (+)

Assignment statement (=)

Scalar relational operator (<, <=, ==, /=, >, >=, .LT., .LE., .EQ., .NE., .GT., .GE.)

Logical operator (.NOT., .AND., .OR., .EQV., .NEQV.)

break ['*file*'] *line*

Sets a breakpoint at the line number *line* in the source file *file*. If omitted, *file* defaults to the current file. Note that the “apostrophes” in ‘file’, above, are the standard apostrophe character (ascii 39).

break ['*file*'] *funcname*

Sets a breakpoint at the entry point of the function *funcname* in the source file *file*. If omitted, *file* defaults to the current file. Note that the “apostrophes” in ‘file’, above, are the standard apostrophe character (ascii 39).

break **addr*

Sets a breakpoint at address *addr*.

break

Sets a breakpoint at the current line.

condition #n expr

Associate conditional expression *expr* with the breakpoint whose serial number is *n*. Note that the “#” symbol is required.

condition #n

Remove any condition associated with the breakpoint whose serial number is *n*. Note that the “#” symbol is required.

oncebreak

Sets a temporary breakpoint that is deleted after the program is stopped at the breakpoint once. OnceBreak in other regards, including arguments, works like Break.

regularbreak "regex"

Set a breakpoint at the beginning of all functions or procedures with a name matching regular expression *regex*.

delete location

Removes the breakpoint at location *location* as described in above syntax description.

delete [' file'] line

Removes the breakpoint for the line number *line* in the source file specified as *file*. If omitted, *file* defaults to the current file. Note that the “apostrophes” in ‘file’, above, are the standard apostrophe character (ascii 39).

delete [' file'] funcname

Removes the breakpoint for the entry point of the function *funcname* in the source file *file*. If omitted, *file* defaults to the current file. Note that the “apostrophes” in ‘file’, above, are the standard apostrophe character (ascii 39).

delete *addr

Removes the breakpoint for the address *addr*.

delete #n

Removes breakpoint number *n*.

delete

Removes all breakpoints.

skip #n count

Skips the breakpoint number *n count* times.

onstop #n cmd[;cmd2;cmd3...;cmdn]

Upon encountering breakpoint *n*, execute the specified fdb command(s).

show break**B**

Displays all breakpoints. If using the abbreviation “B”, the “B” must be upper case.

Controlling Program Execution

continue [*count*]

Continues program execution until a breakpoint's count reaches *count*. Then, execution stops. If omitted, count defaults to 1 and the execution is interrupted at the next breakpoint. Program execution is continued without the program being notified of a signal, even if the program was broken by that signal. In this case, program execution is usually interrupted later when the program is broken again at the same instruction.

silentcontinue [*count*]

Same as Continue but if a signal breaks a program, the program is notified of that signal when program execution is continued.

step [*count*]

Executes the next *count* lines, including the current line. If omitted, *count* defaults to 1, and only the current line is executed. If a function or subroutine call is encountered, execution “steps into” that procedure.

silentstep [*count*]

Same as Step but if a signal breaks a program, the program is notified of that signal when program execution is continued.

step*i* [*count*]

Executes the next *count* machine language instructions, including the current instruction. If omitted, *count* defaults to 1, and only the current instruction is executed.

silentstep*i* [*count*]

Same as Step*i* but if a signal breaks a program, the program is notified of that signal when program execution is continued.

next [*count*]

Executes the next *count* lines, including the current line, where a function or subroutine call is considered to be a line. If omitted, *count* defaults to 1, and only the current line is executed. In other words, if a function or subroutine call is encountered, execution “steps over” that procedure.

silentnext [*count*]

Same as Next but if a signal breaks a program, the program is notified of that signal when program execution is continued.

nexti [*count*]

Executes the next *count* machine language instructions, including the current instruction, where a function call is considered to be an instruction. If omitted, *count* defaults to 1, and only the current instruction is executed.

silentnexti [*count*] or nin [*count*]

Same as Nexti but if a signal breaks a program, the program is notified of that signal when program execution is continued.

until

Continues program execution until reaching the next instruction or statement.

until loc

Continues program execution until reaching the location or line *loc*.

until *addr

Continues program execution until reaching the address *addr*.

until +/-offset

Continues program execution until reaching the line forward (+) or backward (-) *offset* lines from the current line.

until return

Continues program execution until returning to the calling line of the function that includes the current breakpoint.

Displaying Program Stack Information

traceback [*n*]

Displays subprogram entry points (frames) in the stack, where *n* is the number of stack frames to be processed from the current frame.

frame [#*n*]

Select stack frame number *n*. If *n* is omitted, the current stack frame is selected. Note that the “#” symbol is required.

upside [*n*]

Select the stack frame for the procedure *n* levels up the call chain (down the chain if *n* is less than 0). The default value of *n* is 1.

downside [*n*]

Select the stack frame for the procedure *n* levels down the call chain (up the chain if *n* is less than 0). The default value of *n* is 1.

show args

Display argument information for the procedure corresponding to the currently selected frame

show locals

Display local variables for the procedure corresponding to the currently selected frame

show reg [\$*r*]

Displays the contents of the register *r* in the current frame. *r* cannot be a floating-point register. If \$*r* is omitted, the contents of all registers except floating-point registers are displayed. Note that the \$ symbol is required.

show freg [\$*fr*]

Displays the contents of the floating-point register *fr* in the current frame. If \$*fr* is omitted, the contents of all floating-point registers are displayed. Note that the \$ symbol is required.

show regs

Displays the contents of all registers including floating-point registers in the current frame.

show map

Displays the address map.

Setting and Displaying Program Variables

set *variable* = *value*

Sets *variable* to *value*.

set *addr = value

Sets *addr to value.

set reg = value

Sets reg to value. reg must be a register or a floating-point register.

print [:F][variable]

Displays the content of the program variable *variable* by using the edit format *F*. If edit format *F* is omitted, it is implied based on the type of variable. *variable* can be a scalar, array, array element, array section, derived type, or derived type element. *F* can have any of the following values:

- x hexadecimal
- d signed decimal
- u unsigned decimal
- o octal
- f floating-point
- c character
- s character string
- a address of variable (use "&variable" to denote l-value)

memprint [:FuN] addr

dump [:FuN] addr

Displays the content of the memory address *addr* by using edit format *F*. *u* indicates the display unit, and *N* indicates the number of units. *F* can have the same values as were defined for the Print command variable *F*.

If omitted, *f* defaults to x (hexadecimal).

u can have any of the following values:

- b one byte
- h two bytes (half word)
- w four bytes (word)
- l eight bytes (long word/double word)

If *u* is omitted, it defaults to w (word). If *n* is omitted, it defaults to 1. Therefore, the two following commands have the same result:

memprint addr
memprint :xw1 addr

Source File Display

show source

Displays the name of the current file.

!list now

Displays the current line.

!list [next]

Displays the next 10 lines, including the current line. The current line is changed to the last line displayed.

!list previous

Displays the last 10 lines, except for the current line. The current line is changed to the last line displayed.

!list around

Displays the last 5 lines and the next 5 lines, including the current line. The current line is changed to the last line displayed.

!list ['file'] num

Changes from the current line of the current file to the line number *num* of the source file *file*, and displays the next 10 lines, including the new current line. If *file* is omitted, the current file is not changed.

!list +/-offset

Displays the line forward (+) or backward (-) *offset* lines from the current line. The current line is changed to the last line displayed.

!list ['file'] top,bot

Displays the source file lines between line number *top* and line number *bot* in the source file *file*. If *file* is omitted, it defaults to the current file. The current line is changed to the last line displayed.

!list [func[ti]on] funcname

Displays the last 5 lines and the next 5 lines of the entry point of the function *funcname*.

disas

Displays the current machine language instruction in disassembled form.

disas *addr1 [,*addr2]

Displays the machine language instructions between address *addr1* and address *addr2* in disassembled form. If *addr2* is omitted, it defaults to the end of the current function that contains address *addr1*.

disas funcname

Displays all instructions of the function *funcname* in disassembled form.

Automatic Display

screen [:F] expr

Displays the value of expression *expr* according to format *F* every time the program stops.

screen

Displays the names and values of all expressions set by the `screen [:F] expr` command above.

unscreen [#n]

Remove automatic display number *n* (“#” symbol required). When *#n* is omitted, all are removed.

screenoff [#n]

Deactivate automatic display number *n*. When *#n* is omitted, all are deactivated.

screenon [#n]

Activate automatic display number *n*. When *#n* is omitted, all are activated.

show screen

Displays a numbered list of all expressions set by the `screen [:F] expr` command above.

Symbols

show function ["regex"]

Display the type and name of all functions or subroutines with a name that matches regular expression *regex*. When *regex* is omitted, all procedure names and types are displayed.

show variable ["*regex*"]

Display the type and name of all variables with a name that matches regular expression *regex*. When *regex* is omitted, all variable names and types are displayed.

Scripts

alias *cmd* "*cmd-str*"

Assigns the fdb command(s) in *cmd-str* to alias *cmd*.

alias [*cmd*]**show alias [*cmd*]**

display the alias *cmd* definition. When *cmd* is omitted, all the definitions are displayed.

unalias [*cmd*]

Remove the alias *cmd* definition. When *cmd* is omitted, all the definitions are removed.

Signals

signal *sig* *action*

Behavior *action* is set for signal *sig*. Please refer to signal(5) for the name which can be specified for *sig*. The possible values for *action* are:

stopped	Execution stopped when signal <i>sig</i> encountered
throw	Execution not stopped when signal <i>sig</i> encountered

show signal [*sig*]

Displays the set response for signal *sig*. If *sig* is omitted, the response for all signals is displayed.

Miscellaneous Controls

param listsize *num*

The number of lines displayed by the `list` command is set to *num*. The initial (default) value of *num* is 10.

param prompt "*str*"

`str` is used as a prompt character string. The initial (default) value is "`fdb*`". Note that the double quotes are required.

param printelements *num*

Set the number of displayed array elements to *num* when printing arrays. The initial (default) value is 200. The minimum value of *num* is 10. Setting *num* to 0 implies no limit.

param *prm*

Display the value of parameter *prm*.

Files

show exec

Display the name of the current executable file.

param execpath [*path*]

Add *path* to the execution file search path. If *path* is omitted, the value of the search path is displayed. Note that this search path is comprised of a list of directories separated by semicolons.

param srcpath [*path*]

Add *path* to the source file search path when searching for procedures, variables, etc. If *path* is omitted, the value of the search path is displayed. Note that this search path is comprised of a list of directories separated by semicolons.

show source

Display the name of the current source file.

show sources

Display the names of all source files in the program.

Fortran 95 Specific

breakall *mdl*

Set a breakpoint in all Fortran procedures (including internal procedures) in module *mdl*.

breakall *func*

Set a breakpoint in all internal procedures in procure *func*.

show ffile

Displays information about the files that are currently open in the Fortran program.

show fopt

Display the runtime options specified at the start of Fortran program execution.

Communicating with fdb**Functions**

In a Fortran 95 program, if modules and internal subprograms are used, functions are specified as the following:

A module subprogram *sub* defined inside a module *module* is specified as *module@sub*.

An entry point *ent* defined inside a module *module* is specified as *module@ent*.

An internal subprogram *insub* defined inside a module subprogram *sub* within a module *module* is specified as *module@sub@insub*.

An internal subprogram *insub* defined inside a subprogram *sub* is specified as *sub@insub*.

The name of the top level function, MAIN_, is not needed when specifying a function.

Variables

Variables are specified in fdb in the same manner as they are specified in Fortran 95 or C.

In C, a structure member is specified as *variable.member* or *variable->member* if *variable* is a pointer. In Fortran 95, a derived-type (i.e., structure) component is specified as *variable%member*.

In C, an array element is specified as *variable [member] [member] ...*. In Fortran 95, an array element is specified as *variable (member , member , ...)*. Note that in Fortran 95, omission of array subscripts implies a reference to the entire array. Listing of array contents in Fortran 95 is limited by the `printelements` parameter (see "Miscellaneous Controls" on page 77).

Values

Numeric values can be of types integer, real, unsigned octal, or unsigned hexadecimal. Values of type real can have an exponent, for example `3.14e10`.

In a Fortran 95 program, values of type complex, logical, and character are also allowed. Values of type complex are represented as *(real-part,imaginary-part)*. Character data is represented as " *character string* " (the string is delimited by quotation marks, i.e., ascii 34).

Values of type logical are represented as `.t.` or `.f.`

Addresses

Addresses can be represented as unsigned decimal numbers, unsigned octal numbers (which must start with 0), or unsigned hexadecimal numbers (which must start with 0x or 0X). The following examples show print commands with address specifications.

memprint 1024 (The content of the area addressed by 0x0400 is displayed.)

memprint 01024 (The content of the area addressed by 0x0214 is displayed.)

memprint 0x1024 (The content of the area addressed by 0x1024 is displayed.)

Registers

\$BP	Base Pointer
\$SP	Stack Pointer
\$EIP	Program counter
\$EFLAGS	Processor state register
\$ST[0-7]	Floating-point registers

Names

In Fortran 95 programs, a lowercase letter in the name (such as a function name, variable name, and so on) is the same as the corresponding uppercase letter. The main program name is MAIN_ and a subprogram name is generated by adding an underscore(_) after the corresponding name specified in the Fortran source program. A common block name is also generated by adding an underscore (__) after the corresponding name specified in the Fortran source program.

5

Windows Debugging with WinFDB

WinFDB is the Windows version of the FDB symbolic source-level debugger for Fortran 95, C, and assembly programs. While debugging, you can watch the values of variables change during program execution and set breakpoints with a mouse click. The WinFDB debugger can run your program, set breakpoints, step a line at a time, view the values of variables in your program in several different ways, and change the values of variables while running your program. The current executable line and any breakpoints are indicated with markers in the left margin of the source code display.

Before debugging your program you must compile it using the `-g` switch (see “*Compiler and Linker Switches*” on page 17). The `-g` switch creates an additional file with debugging information -- this file has the same name as the executable with the extension `.ydg`. Debugging cannot be performed without the presence of the `.ydg` file in the same directory as the executable file. WinFDB cannot be used on LF90 executables.

This chapter assumes a basic familiarity with Windows. It presents an overview of WinFDB’s functionality. More detailed information is available through WinFDB’s on-line help.

How to Start and Terminate WinFDB

There are three ways to start the WinFDB debugger:

1. From the Windows command prompt
2. From the desktop icon or from the Start menu
3. From the Lahey ED developer

Starting from the command prompt

Type WINFDB followed optionally by the name of the executable file to be debugged:

WINFDB [*filename*]

Unless the full path of *filename* is provided, WinFDB will assume it resides in the current working directory.

Starting from the Windows desktop

Start the debugger by double-clicking the WinFDB icon if it is present (the desktop icon is offered as an option at installation time); otherwise use the Start | Programs... dialog.

Starting from the ED Developer

There are two ways of starting the WinFDB debugger while in ED. You can click on the Debug button from the toolbar, or select `Tools | Debug`. ED will assume the executable file has the same name as the source file in the currently active edit window.

If prompted, enter the name of the executable file, including the filename extension (`.exe`) and, if the file is not in ED's current directory, a path. For example, if the executable file `myprog.exe` is in a directory called "programs" below the root, you would enter

```
\programs\myprog.exe
```

Terminating the Debugger

Terminate the Debugger by selecting the Exit Debugger command from the File menu in the debugger window.

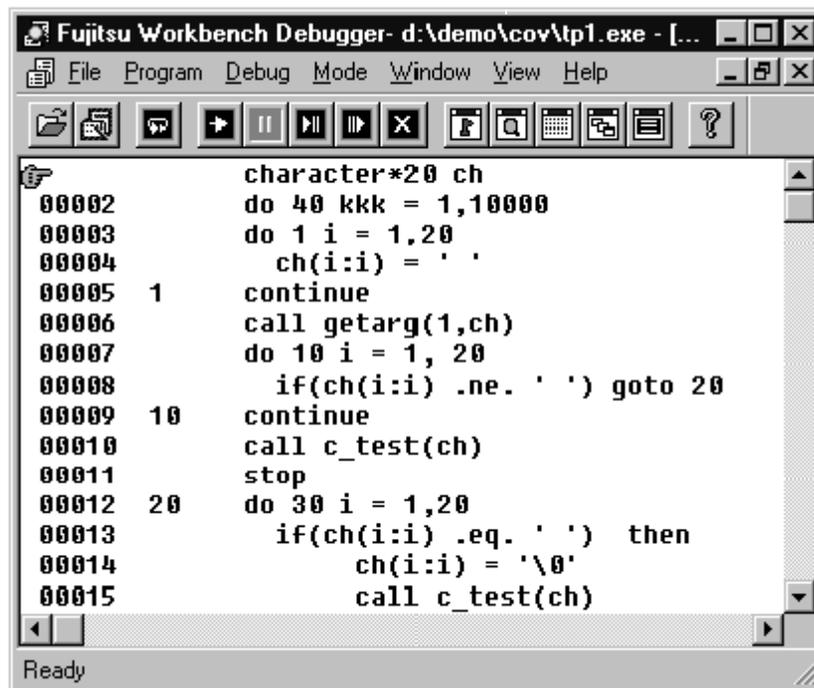
Debugger Window

See the figure below.

Menu bar

Tool bar

Current line
of execution



Status bar

Debugger Window

The following items are visible in the above figure:

1. The source code display, which shows the the lines of source code corresponding to the section of the program currently being executed. The left hand margin of this display shows the line numbers of the code, along with symbols indicating the current line of execution and breakpoints, if any.
2. The Menu bar, used for activating all WinFDB commands.
3. The toolbar, which contains icons that activate frequently used menu commands. These commands can also be executed from the Menu Bar.
4. The status bar, which displays an explanation of each menu or toolbar command.

Debugger Menus

File Menu

The table below lists the File Menu commands:

Table 9: File Menu Commands

Command Name	Function
Open	Selects executable file to be debugged
Exit Debugger	Terminates the WinFDB debugger

Program Menu

The table below lists the Program Menu commands.

Table 10: Program Menu Commands

Command Name	Function
Restart	Reruns the same program.
Set Options	Specifies the argument(s) of the program to be debugged and the runtime option(s) at execution (i.e., command-line arguments)

Debug Menu

The table below lists the Debug Menu commands.

Table 11: Debug Menu commands

Command Name	Function
Go	Runs the program to be debugged; continues an execution that stopped due to a breakpoint, etc.
Interrupt	Pauses the execution of the visual step mode.
Step In	Runs the next statement. Runs up to the beginning of the function if the statement includes the function (i.e., “step into”).
Step Over	Runs up to the next line, assuming the function call is one line
Kill	Stops the debugging session
Breakpoints...	For displaying, setting, and clearing breakpoints.
Watch...	For selecting and displaying variables during execution.
Registers	Displays current values of CPU registers.
Traceback	Displays the traceback information
Map	Displays the modules currently loaded in memory.
Input Command...	Enter FDB commands for detailed debugging; the results appear in the Input Command Log window. See <i>FDB”Commands”</i> on page 67

Mode Menu

The table below lists the Mode Menu commands.

Table 12: Mode Menu Commands

CommandName	Function
Visual Step	Activates the visual step mode

Window Menu

The table below lists the Window Menu commands (note - the Window menu is displayed only if one or more of the Debugger's child windows are displayed):

Table 13: Window Menu commands

Command Name	Function
Cascade	Displays all open windows so that they overlap, revealing the Title Bar for each window.
Tile Horizontally	Displays debug information from left to right.
Tile Vertically	Displays debug information from top to bottom.
Arrange Icons	Arranges all the icons along the bottom of the window.
Close All	Close all open windows

View Menu

The table below lists the View Menu commands:

Table 14: View Menu commands

Command Name	Function
Toolbar	Specifies whether to display the toolbar.
Status Bar	Specifies whether to display the status bar.

Help Menu

The table below lists the Help Menu commands:

Table 15: Help Menu commands

Command Name	Function
Help Topics	Displays the help topics.
About Debugger	Displays version information for the debugger.

Using the Debugger

The debugger has the following functions:

- Starting the program to be debugged
- Setting and deleting breakpoints
- Running and stopping the program
- Displaying debug information

Starting the Program

The first step in debugging is to ensure that the program to be debugged is loaded into the debugger. If the debugger is invoked from the command line or the ED Developer with an executable file specified, the file is loaded automatically. If the executable file is not specified, follow these steps:

1. Click the `Open` command in the `File` menu to display the `Open File` dialog box.
2. In the `Open File` dialog box, click or double-click the program to be debugged (executable file).
3. Click the `Open` button.

After you complete these steps, start debugging by clicking the `Go` command in the `Debug` menu twice; once to load the executable, and once more to begin the debug session. The source program window will open, and the finger icon will appear at the program starting point. If the main program is compiled without specifying the `-g` option, the source program and the finger icon are not displayed.

Setting and Deleting Breakpoints

Setting a Breakpoint

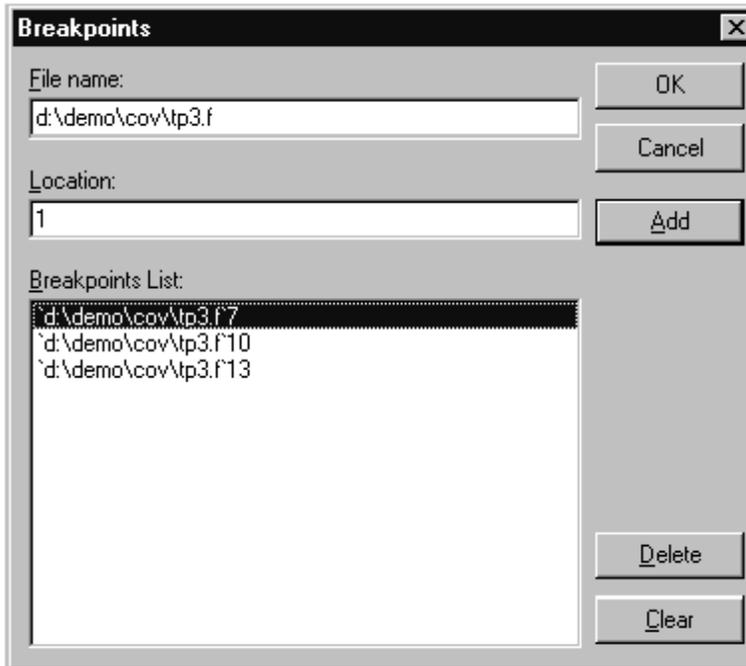
You can stop program execution at any point by setting a breakpoint, allowing the status of variables and registers to be examined. A breakpoint can only be set on an executable line of code; it cannot be set at a comment or variable declaration, for example.

Start the program to be debugged and set the breakpoint when debugging enters start status. To set a breakpoint at a line, position the mouse pointer on the line number in the source program and click the left mouse button.

A breakpoint can also be set as follows:

1. Select the `Breakpoints` command from the `Debug` menu to display the `Breakpoints` dialog box (see the figure below).
2. In the `Position` field in the `Breakpoints` dialog box, specify the line number for which the breakpoint is to be set.

3. Click the Add button.
4. Check that the line number appears in the breakpoint list; then click the OK button.
5. The above step displays the breakpoint (flag) in the displayed source program. To set the breakpoint for another line number, repeat steps (2) and (3).



Releasing the Breakpoint

Some or all breakpoints that have been set can be deleted.

To delete a breakpoint at a line, position the pointer on the line number in the source program (indicated with a flag) and click the left button.

A breakpoint can also be deleted as follows:

1. Select the Breakpoints command from the Debug menu to display the Breakpoints dialog box.
2. In the Breakpoints List field in the Breakpoints dialog box, click the line number to be deleted.
3. Click the Delete button, then the OK button.

All breakpoints can be deleted as follows:

1. Select the `Breakpoints` command from the `Debug` menu to display the `Breakpoints` dialog box.
2. Click the `Clear` button in the `Breakpoints` dialog box, then click the `OK` button.

Running and Stopping the Program

Running the Program

To run the program until the first (or next) breakpoint, select the `Go` command from the `Debug` menu. To step one line, entering a function call if present (“step into”), select the `Step` command from the `Debug` menu. To step one line and treat a function call as one instruction (“step over”), select the `Next` command from the `Debug` menu.

You can execute the program in “Visual Step Mode” by using the `Visual Step` command in the `Mode` menu. Visual Step Mode allows you to run the program “slow motion”, seeing each step as it is executed, and it works as follows:

When you select the `Go` command from the `Debug` menu in the visual step mode, the finger icon moves line by line as the program is executed. This provides a means of visually checking the program execution sequence. To pause the execution in the visual step mode, select the `Interrupt` command from the `Debug` menu.

Stopping the Program

To stop the program, select the `Kill` command from the `Debug` menu. The “debugging enabled” status is released.

To restart debugging, see below.

Rerunning the Program

To restart debugging, reload the program by selecting the `Restart` command from the `Program` menu.

Then, start the program by selecting the `Go` command from the `Debug` menu.

Displaying Debug Information

The debugger can display the following debug information:

- Variables
- Registers
- Traceback
- Load map
- Output

The displayed debug information is updated at the following times:

- When a step run or line run is executed.
- When a program is stopped at a breakpoint.
- When a program running in the visual step mode is paused.

The display method for each type of debug information is listed below.

Displaying Variables

Do the following to display the contents of variables:

1. Select the Watch command from the Debug menu to display the Watch dialog box (see the figure below).
2. In the Variable field, specify the variable to be displayed.
3. Click the Add button. The specified variable is then registered in the variable field. When the program is executed, the current variable contents are displayed on the right-hand side of the “=” symbol.

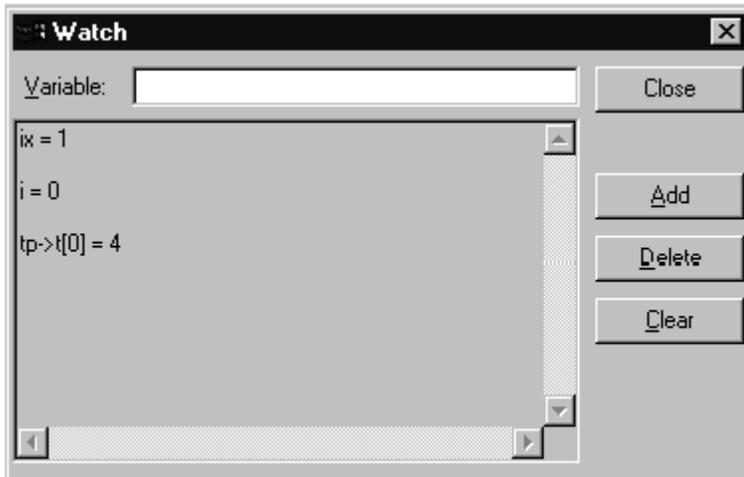
Variable contents are not displayed when any of the following is true:

- The program to be debugged has not been started.
- The local variable of another function or routine was specified.
- A non-used variable was specified.

To delete the registered variable, do one of the following:

1. In the Variable field, specify the variable name to be deleted and click the Delete button.
2. Click the registered variable, then click the Delete button. If you click the Delete button without clicking a variable, the first variable in the variable field is deleted.

To delete all the variables registered in the variable field, click the Clear button.

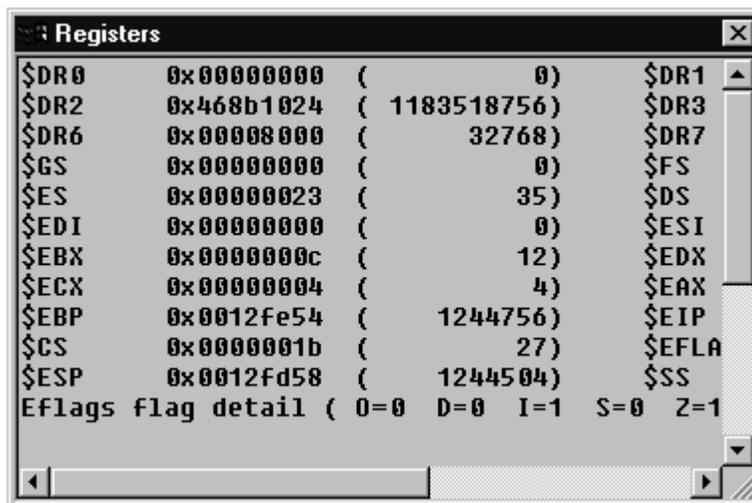


Displaying Registers

Do the following to display the register contents:

1. Select the Registers command from the Debug menu.

The current register contents are displayed in the Registers window (see the figure below).

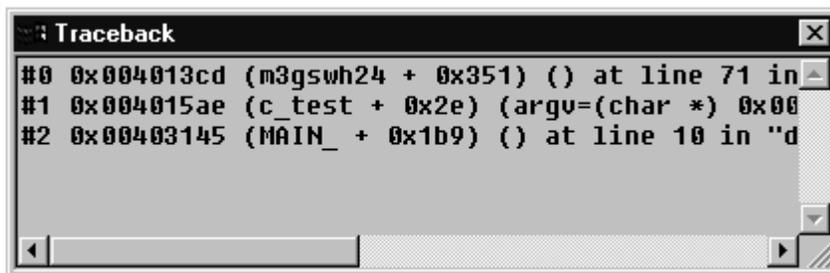


Displaying a Traceback

Do the following to display a traceback:

1. Select the Traceback command from the Debug menu.

The current traceback information is displayed in the Traceback window (see the figure below).

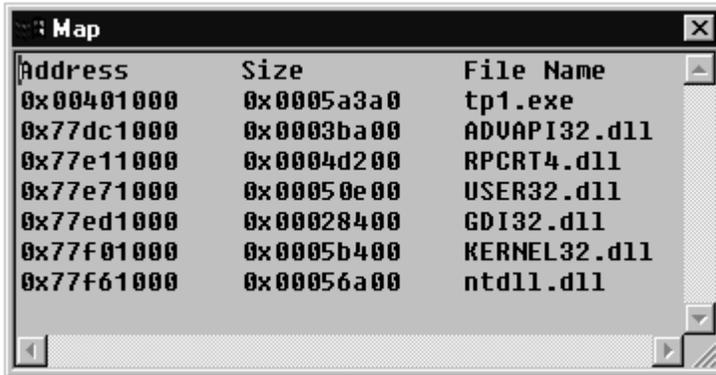


Displaying a Load Map

Do the following to display a load map:

1. Select the Map command from the Debug menu.

The current loaded modules are displayed in the Map window (see the figure below).

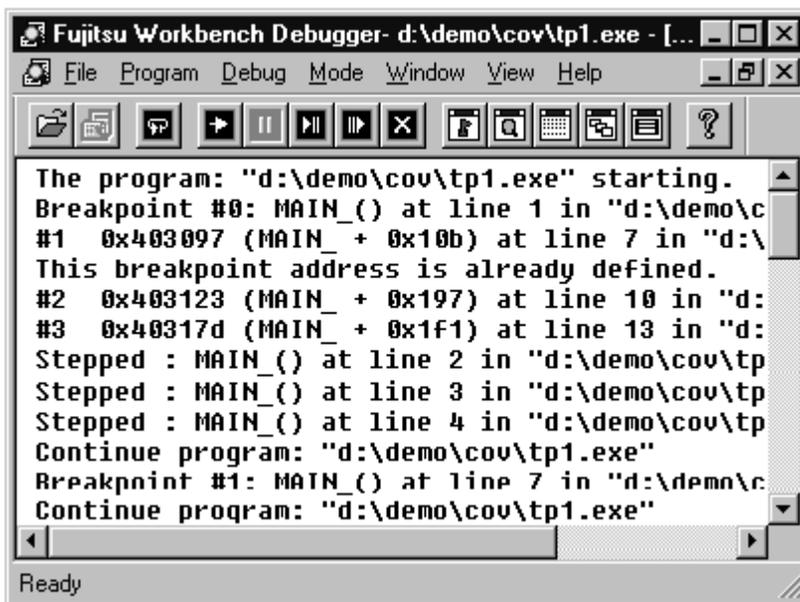


Entering FDB Commands

Do the following to enter FDB commands for more detailed debugging activity:

1. Select the "Input Command" command from the Debug menu.
2. Type a command in the "Command" field in the "Input Command" dialog, or select a command from the drop-down command history, or simply click OK to repeat the last command (see FDB "Commands" on page 67).

The messages and results are displayed in the Input Command Log window (see the figure below).



Restrictions

1. The Debugger will not debug a program that has been compiled by another vendor's compiler, even if their debug option is specified. Other vendor's debuggers will not debug a program that has been compiled by LF95, even if the debug option (-g) is specified.
2. You can combine objects compiled by the Fujitsu Compiler with objects compiled by certain other compilers (see "Mixed Language Programming" on page 38). However, WinFDB will not step into those parts of the programs which were compiled by the other vendor's compiler.
3. An adjustable array that is a dummy argument cannot be debugged if it appears at the beginning of a procedure or a function.

Example:

```

line number
1      subroutine sub(x,y,i)
2      real x(5:i)
3      real y(i+3)

```

In this example, adjustable arrays *x* and *y* cannot be debugged where the beginning of the procedure is sub(line number 1).

4. The dummy argument of a main entry cannot be debugged at the sub entry.

Example:

```
subroutine sub(a,b)
  :
  :
  entry ent(b)
```

In this example, the dummy argument a, which is in the main entry's argument list but not in the sub entry ent, cannot be debugged. However, the dummy argument b, which is passed to the sub entry ent, can be debugged.

5. A breakpoint cannot be set for an executable statement in an include file.

6. An array of an assumed size can be debugged only for the lower boundary.

7. A label cannot be debugged.

8. In include files that contain expressions or a #line statement, the C programs cannot be debugged.

9. When in a Fortran program the continue statement has no instruction, the breakpoint is automatically set at the next statement.

Example:

```
line number
1      integer i
2      :
3      assign 10 to i
4      10 continue
5      i=1
```

In the above example, if you set a breakpoint at line 4, the breakpoint is actually set at line 5.

10. If a Fortran program has a contains statement, the breakpoint cannot be set at its end statement.

11. If the result of a function is one of the following, the step and next commands have no effect:

- array
- derived type
- pointer
- character whose length is not a constant

12. An allocated array cannot be debugged except for the first element of the array.

13. If a pointer variable is a common element, the pointer variable cannot be debugged.

Example:

```
common /com/p
pointer (p,j)
```

The above variable `j` cannot be debugged.

14. A dummy argument declared with the entry statement in a procedure cannot be debugged except for the first element of the explicit-shape array and the top of the assumed type parameter string (including the pointervariable).

Example:

```
subroutine sl(a,i,c)
  real a(i)
  character*(*) c
  :
  entry ent(b,j,d)
```

The above cannot be debugged except `a(1)` and `c(1:1)`.

15. When debugging a program using the VSW function, please note that Execution should be used to restart the execution after returning from the call-back routine. If Step or Line is used to restart the execution, breakpoints may not be ignored.

Other Remarks

1. In source level debugging, the executable file (.EXE) and its debugging information file (.YDG) must exist in the same directory. In the same way, the dynamic link library (.DLL) and its debugging information file must exist in the same directory.

2. In source level debugging, the prolog instructions of each function may cause the following features not to work correctly:

- traceback indication
- next command

3. When searching the source files, the Debugger refers to the environment variable `FDB_SRC_PATH`. There are two ways of specifying the environment variable:

Example 1: In the command prompt

```
c:\> set FDB_SRC_PATH=c:\users\fujitsu\prog;d:\common\lib\src
c:\> winfdb
```

The above specifies the full pathnames of the directory in which the source files exist. If there are more than two directories, you may specify them with the separator ";". Then, invoke the Debugger.

Example 2: In the Control Panel (Windows NT only)

```
Variable(V): FDB_SRC_PATH
Value(A): c:\users\fujitsu\prog;d:\common\lib\src
```

The above specifies environment variable `FDB_SRC_PATH` to each user's environment variable of the System in the Control Panel. In Windows 95/98, specify the environment variable `FDB_SRC_PATH` in `AUTOEXEC.BAT`.

4. If the debug option is specified when linking, object filenames must be specified with full pathnames except for the objects in the current directory.

5. If objects are linked with the debug option and static link libraries are in a different directory from the object files and debugging information files(.YDG), to debug the executable file, specify the full path-names of the object files and the debugging information file to the environment variable FDB_MERG_PATH.

Example: Specify the environment variable FDB_MERG_PATH

```
set FDB_MERG_PATH=c:\users\fujitsu;e:\apl\users\lib\obj
```

In the above, the directory name is specified with full pathnames separated by a semicolon.

Note: The object file and debugging information file are searched as follows:

1. The directories specified in the environment variable FDB_MERG_PATH.
2. The directories which store the user's library.

6

LM Librarian

The Lahey Librarian, LM, can be used to create, modify, and inspect object library files.
Command-Line Syntax

```
LM [old-library-name][switches] [commands] [, [list-filename][,new-library-name];
```

Where:

old-library-name is the name of an existing library used as input.

switches is zero or more command-line switches (described below).

commands is zero or more commands (described below).

list-filename is the name of the listing file.

new-library-name is the name of a new library to create if you do not want to update the old library.

A file name can be specified on the command line as a complete file name (*filename .ext*) or can be given without an extension, in which case 386LIB supplies a default extension of `.obj` for an object file and `.lib` for a library. There is no default extension for a listing file. In addition, a complete or partial path may be specified with the file name. If none is given, the current default device and directory are assumed.

There should be no delimiter between multiple commands or switches.

Switches

/EXTRACTALL

The `/EXTRACTALL` switch extracts all object modules from a specified library. Each extracted object module becomes an object file.

Example

```
LM mylib/extractall;
```

/PAgesize

The /PAgesize switch allows you to specify the page size (in bytes) of a library. Specifying a bigger page size will enable you to store more objects. A smaller page size will result in a smaller library. The default page size is either 16 bytes or the page size of the existing library. Valid page sizes are integral powers of 2 from 16 to 32768, inclusive. The syntax for /PAgesize is:

```
/PA[gesize]:page-size
```

where *page-size* is the page size in bytes.

Example

```
LM mylib/pa:16384;
```

/Help

The /Help switch prints a summary of syntax and usage for the library manager. The syntax for /Help is:

```
/H[elp]
```

Example

```
LM/H;
```

Commands

Add Modules

The addition symbol (+) appends the object file(s) following each '+' symbol to the end of the library. Pathnames can be used if the object file is not in the current directory.

Example

```
LM mylib +myobj;
```

The object `myobj.obj` is added to `mylib.lib`.

Delete Modules

The subtraction symbol (-) preceding an object module name deletes that module from the library specified with *old-library-name*.

Example

```
LM mylib -myobj;
```

The object module `myobj` is deleted from `mylib.lib`.

Replace Modules

The subtraction and addition symbols (`-+`) combine to form the Replace Modules command. The Replace Modules command, followed by an object module name, replace that module by deleting it from the library and then appending an object file with the same name as the deleted module to the library.

Example

```
LM mylib -+myobj-+myobj2;
```

The object modules `myobj` and `myobj2` are replaced with `myobj.obj` and `myobj2.obj`.

Copy Modules

The asterisk symbol (`*`) copies the module following the asterisk into an object file with the same name.

Example

```
LM mylib *myobj;
```

The object module `myobj` from `mylib.lib` is copied to `myobj.obj`. The object module `myobj` in the library is unaffected.

Move Modules

The subtraction symbol followed by an asterisk (`-*`) forms the Move Modules command, which moves a specified object module to an object file of the same name.

Example

```
LM mylib -*myobj-*myobj2;
```

The object modules `myobj` and `myobj2` are deleted from `mylib.lib` and `myobj.obj` and `myobj2.obj` are created.

Response Files

It is possible to place commonly used or long LM command-line parameters in a response file. LM command-line parameters are entered in a response file in the same manner as they would be entered on the command line. A new line in a response file is treated like a comma

on the LM command line. The ampersand character (&) is used to continue a line in a command file. The ampersand character is valid only after an object module name or object filename. A blank line is interpreted as the end of input to LM.

To invoke the response file, type:

LM @*response-filename*

where *response-filename* is the name of the response file.

Example

```
oldlib
+object+object2-module3&
*module4
listfile.lst
newlib
```

This adds the objects `object` and `object2` to what was in `oldlib.lib`, removes `module3`, and copies `module4` to `module4.obj`. The listing file is written to `listfile.lst` and the new library is called `newlib.lib`.

Interactive Mode

LM can be operated in interactive mode by typing just LM at the command prompt. You will be prompted for input. The ampersand character is used to continue a long line of input. The following session would perform the same operations as the command file above:

Example

```
lm
Library name: oldlib
Operations desired: +object+object2-module3&
Operations desired: *module4
List filename: listfile.lst
Output library name: newlib
```



Automake

Introduction

What Does It Do?

AUTOMAKE is a simple-to-use tool for re-building a program after you have made changes to the Fortran and/or C source code. It examines the creation times of all the source, object and module files, and recompiles wherever it finds that an object or module file is non-existent, empty or out of date. In doing this, it takes account not only of changes or additions to the source code files, but also changes or additions to MODULEs and INCLUDED files - even when nested. For example, if you change a file which is INCLUDED in half a dozen source files, AUTOMAKE ensures that these files are re-compiled. In the case of Fortran 90, AUTOMAKE ensures that modules are recompiled from the bottom up, taking full account of module dependencies.

How Does It Do That?

AUTOMAKE stores details of the dependencies in your program (e.g. file A INCLUDEs file B) in a dependency file, usually called 'automake.dep'. AUTOMAKE uses this data to deduce which files need to be compiled when you make a change. Unlike conventional MAKE utilities, which require the user to specify dependencies explicitly, AUTOMAKE creates and maintains this data itself. To do this, AUTOMAKE periodically scans source files to look for INCLUDE and USE statements. This is a very fast process, which adds very little to the overall time taken to complete the update.

How Do I Set It up?

The operation of AUTOMAKE is controlled by a configuration file which contains the default compiler name and options, INCLUDE file search rule, etc. For simple situations, where the source code to be compiled is in a single directory, and builds into a single execut-

able, it will probably be possible to use the system default configuration file. In that case there is no need for any customization of AUTOMAKE— just type 'am' to update both your program and the dependency file.

In other cases, you may wish to change the default compiler name or options, add a special link command, or change the INCLUDE file search rule; this can be achieved by customizing a local copy of the AUTOMAKE configuration file. More complex systems, perhaps involving source code spread across several directories, can also be handled in this way.

What Can Go Wrong?

Not much. AUTOMAKE is very forgiving. For example, you can mix manual and AUTOMAKE controlled updates without any ill effects. You can even delete the dependency file without causing more than a pause while AUTOMAKE regenerates the dependency data. In fact, this is the recommended procedure if you do manage to get into a knot.

Running AUTOMAKE

To run AUTOMAKE, simply type 'am'. If there is a configuration file (AUTOMAKE.FIG) in the current directory, AUTOMAKE reads it. Otherwise, it starts the AUTOMAKE Configuration file editor, AMEDIT.EXE.

The AUTOMAKE Configuration File Editor

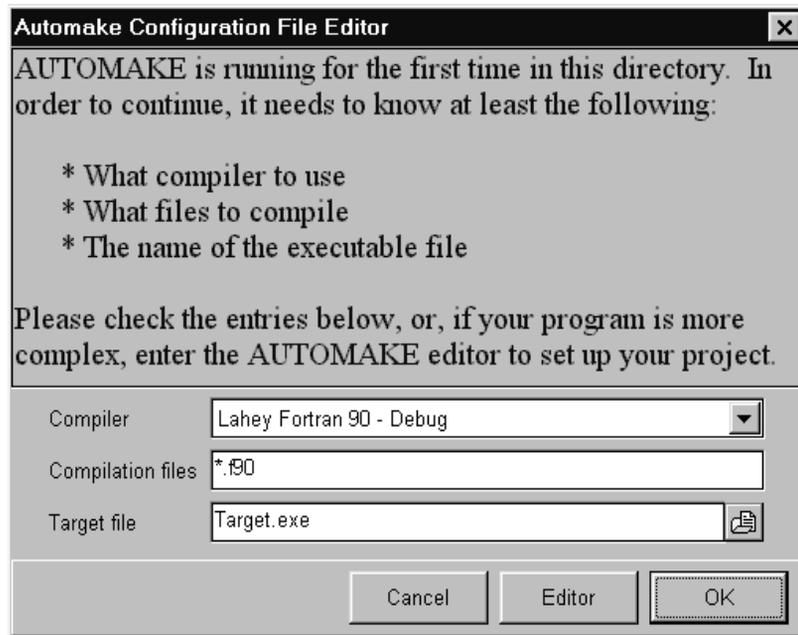
The AUTOMAKE Configuration File Editor (AMEDIT) is a Windows-based utility for creating and maintaining configuration files for use by AUTOMAKE. You can start it from a Windows 95 or NT command prompt by typing

```
amedit
```

to create a new file, or

```
amedit myproject.fig
```

to edit an existing one. AMEDIT is also started automatically when AUTOMAKE first runs in a directory with no AUTOMAKE.FIG file.

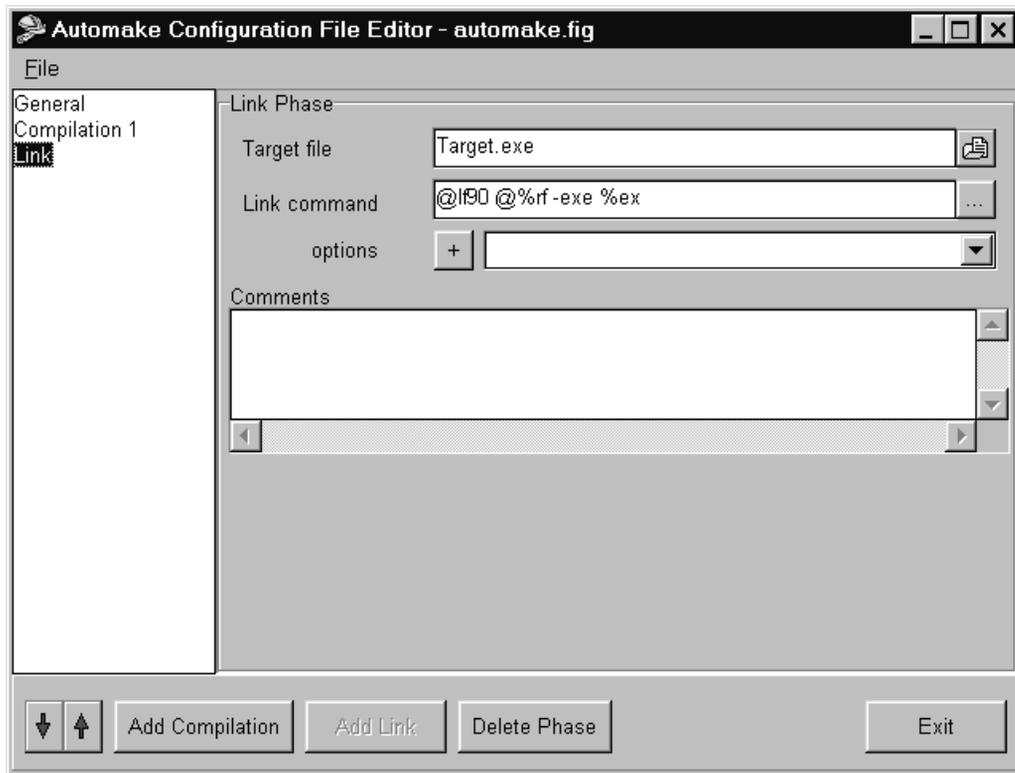


If you are creating a new file, the editor first presents a dialog containing a set of options suitable for most simple projects. The defaults are:

- to use LF95 with switches set for debugging. You can select other options, including LF95 with switches set for optimization, from the drop-down list.
- to compile all files fitting `*.f90` in the current directory. This can be changed, for example to `*.for`, by typing in the second box.
- to create an executable file called `target.exe`. This can be changed by typing in the third box, or by using the file selection dialog (click on the button at the right of the box).

When you are finished, click “OK” to create the file.

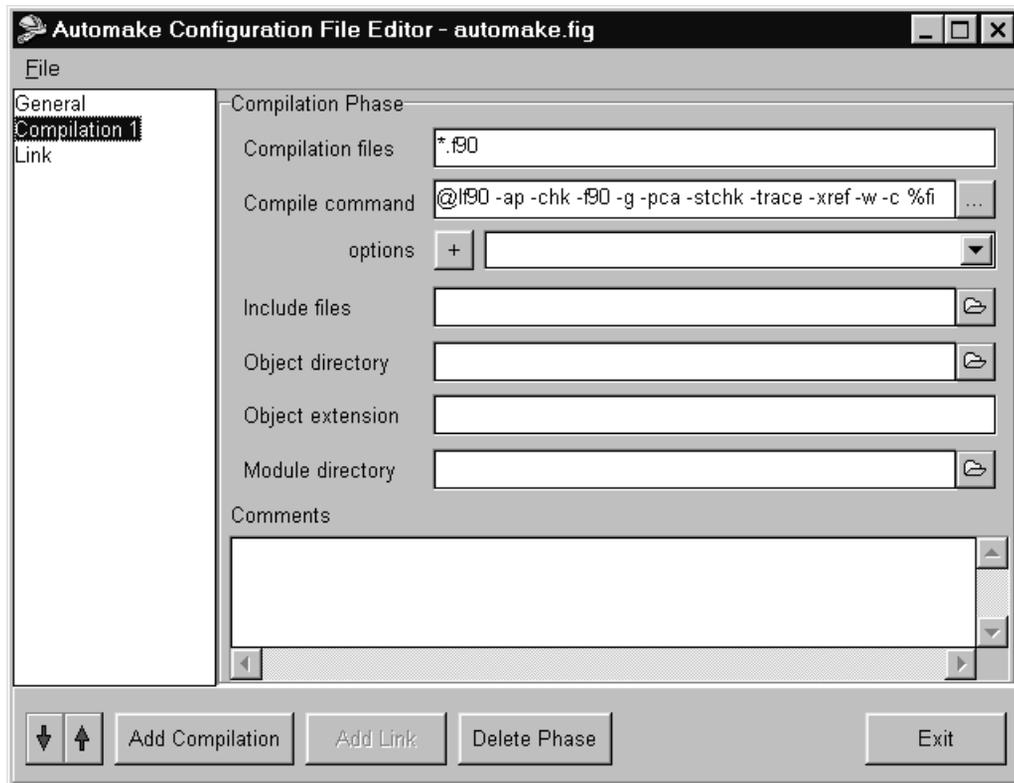
If your project is more complicated than that—for example if you have files in more than one directory, or you need special linker instructions—click on the “Editor” button and a new dialog with many more possibilities is displayed.



The pane on the left-hand side lists the sections in the `AUTOMAKE.FIG` file. When you click on the section in the left-hand pane, details appear in the right hand pane. Typically there is a general section, which specifies options, such as debug switches, which relate to the entire process. One or more compilation sections follow this, each specifying a set of files to be compiled together with the compiler options. Finally, there is usually a link section, in which the link command is specified.

The link section, shown above, allows you to enter the executable file name and the link command (see "`LINK=`" on page 109 for an explanation of place markers such as `%rf` and `%ex`). There is a drop-down list of linker options which, once selected, can be added to the link command by clicking on the '+' button. Finally, you can add comments as required in the box at the bottom of the right-hand pane.

Compilation sections are similar to link sections, but with a few more options:



This time you must specify the files to be compiled (see "*FILES=*" on page 107) and the compile command (see "*COMPILE=*" on page 107). As in the LINK section, there is a drop-down list of compiler options that can be appended to the compile command by clicking on the '+' button.

The other entries are all optional. They are:

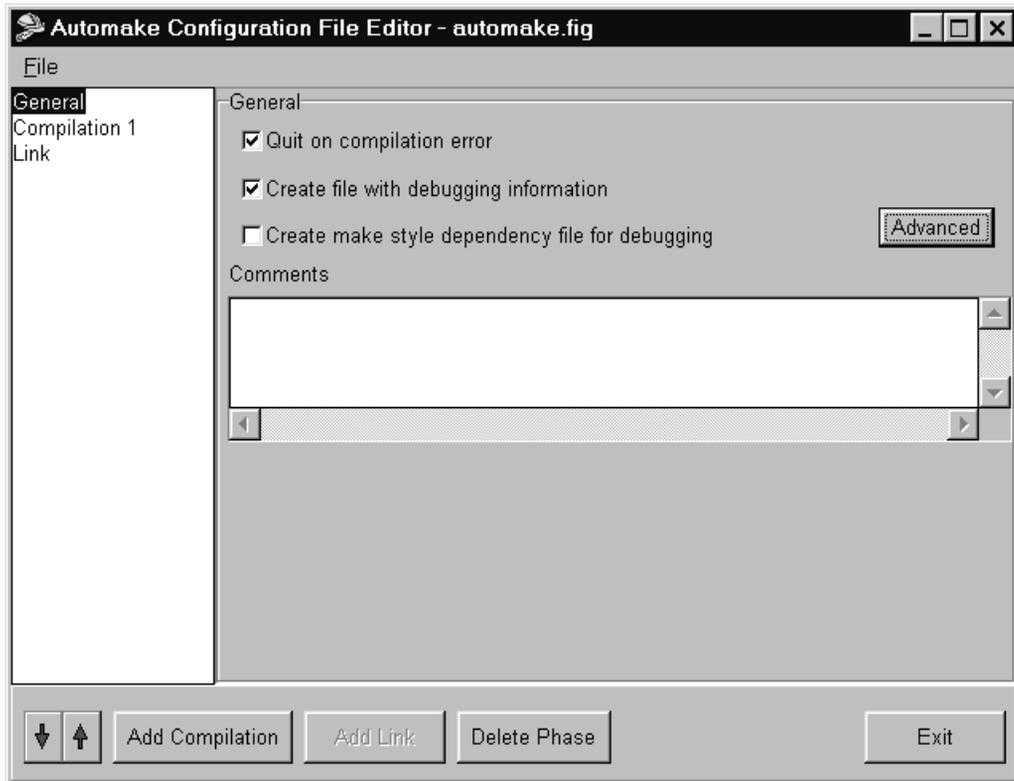
- The directories to be searched for INCLUDE files (see "*INCLUDE=*" on page 110)
- The target directory for object files (see "*OBJDIR=*" on page 110)
- The object file extension (see "*OBJEXT=*" on page 110)
- The target directory for .MOD files (see "*MODULE=*" on page 110)
- Comments

New compilation sections can be added by clicking on "Add Compilation", and deleted by clicking on "Delete Phase". The order of compilation sections can be changed using the arrow buttons at the bottom left.

The General Section includes three check boxes:

- To specify whether AUTOMAKE should continue after a compilation error (see "QUITONERROR" on page 111 and "NOQUITONERROR" on page 111).
- To specify whether debugging data should be written to a file called AUTOMAKE.DBG (see "DEBUG" on page 111).
- To specify whether a simple make file called AUTOMAKE.MAK should be written as an aid to debugging (see "MAKEMAKE" on page 111).

The Advanced button gives access to options that are not relevant for LF95.



The AUTOMAKE Configuration File

The AUTOMAKE configuration file is used to specify the compile and link procedures, and other details required by AUTOMAKE. It consists of a series of records of the form

keyword=value

or

keyword

where *keyword* is an alphanumeric keyword name, and *value* is the string of characters assigned to the keyword. The keyword name may be preceded by spaces if required. Any record with a '#', '!' or '*' as the first non-blank character is treated as a comment.

The keywords which may be inserted in the configuration file are:

LF95

Equivalent to specifying the default LF95 compile and link commands.

```
COMPILE=@lf95 -c %fi -mod %mo
LINK=@lf95 %ob -exe %ex -mod %mo
```

The LF95 keyword should appear in any automake.fig file that is to be used with LF95.

FILES=

Specifies the names of files which are candidates for re-compilation. The value field should contain a single filename optionally including wild-cards. For example,

```
FILES=*.f90
```

You can also have multiple FILES= specifications, separated by AND keywords.

```
FILES=F90\*.F90
AND
FILES=F77\*.FOR
AND
...
```

Note that, with each new FILES= line, the default COMPILE= is used, unless a new COMPILE= value is specified after the FILES= line and before AND.

Note also that, if multiple FILES= lines are specified, then the %RF place marker cannot be used in any COMPILE= lines.

COMPILE=

Specifies the command to be used to compile a source file. The command may contain place markers, which are expanded as necessary before the command is executed. For example,

```
COMPILE=@lf95 -c %fi
```

The string '%fi' in the above example is a place marker, which expands to the full name of the file to be compiled. The following table is a complete list of place markers and their meanings:

Table 16: COMPILER= Place Markers

Place Marker	Meaning
%SD	expands to the name of the directory containing the source file - including a trailing '\'.
%SF	expands to the source file name, excluding the directory and extension.
%SE	expands to the source file extension—including a leading '.'. For example if the file to be compiled is 'f:\source\main.for', %SD expands to 'f:\source\'', %SF to 'main', and %SE to '.for'.
%OD	expands to the name of the directory containing object code, as specified using the OBJDIR= command (see below), including a trailing '\'.
%OE	expands to the object file extension, as specified using the OBJEXT= command (see below), including a leading '.'.
%ID	expands to the INCLUDE file search list (as specified using INCLUDE= (see below))
%MO	expands to the name of directory containing modules (as specified using MODULE= (see below))
%RF	expands to the name of a response file, created by AUTOMAKE, containing a list of source files. If %RF is present, the compiler is invoked only once.
%FI	is equivalent to %SD%SF%SE

```
COMPILER=@lf95 -c %fi -mod %mo
```

```
COMPILER=@lf95 -c @%rf -I %id
```

It is possible to invoke the compiler using a command file (batch file, shell script etc.). However, on PCs, it is necessary to preface the batch file name with 'CALL' or 'COMMAND/C'. For example

```
COMPILER=CALL fcomp %fi
```

Note that with LF95 the -c switch should always be used in a COMPILER= line.

TARGET=

Specifies the name of the program or library file which is to be built from the object code. Note that you will also have to tell the linker the name of the target file. You can do this using a %EX place marker (which expands to the file name specified using TARGET=).

```
TARGET=f:\execs\MYPROG.EXE
```

If there is no TARGET= keyword, AUTOMAKE will update the program object code, but will not attempt to re-link.

LINK=

Specifies a command which may be used to update the program or library file once the object code is up to date:

```
LINK=@lf95 %ob -exe %ex -mod %mo'
```

```
LINK=@lf95 %od*%oe -exe %ex -mod %mo'
```

```
LINK=@lf95 %rf -exe %ex -mod %mo'
```

You could use a batch file called 'l.bat' by specifying

```
LINK=CALL L
```

The following place markers are allowed in the command specified using LINK=.

Table 17: LINK= Place Markers

Place Marker	Meaning
%OD	expands to the name of the directory containing object code, as specified using the OBJDIR= command (see below), including a trailing '\'.
%OE	expands to the object file extension, as specified using the OBJEXT= command (see below), including a leading '.'.
%OB	expands to a list of object files corresponding to source files specified using all FILES= commands.
%EX	expands to the executable file name, as specified using TARGET=.
%MO	expands to the name of directory containing modules (as specified using MODULE= (see below))
%RF	expands to the name of a response file, created by AUTOMAKE, containing a list of object files.

INCLUDE=

May be used to specify the INCLUDE file search list. If no path is specified for an INCLUDED file, AUTOMAKE looks first in the directory which contains the source file, and after that, in the directories specified using this keyword. The directory names must be separated by semi-colons. For example, on a PC, we might have:

```
INCLUDE=C:\include;C:\include\sys
```

Note that the compiler will also have to be told where to look for INCLUDED files. You can do this using a %ID place marker (which expands to the list of directories specified using INCLUDE).

SYSINCLUDE=

May be used to specify the search list for system INCLUDE files (i.e. any enclosed in angled brackets), as in

```
#include <stat.h>
```

If no path is specified, AUTOMAKE looks in the directories specified using this keyword. It does not look in the current directory for system INCLUDE files unless explicitly instructed to. The directory names following SYSINCLUDE= must be separated by semi-colons.

OBJDIR=

May be used to specify the name of the directory in which object files are stored. For example,

```
OBJDIR=OBJ\
```

The trailing '\' is optional. If OBJDIR= is not specified, AUTOMAKE assumes that source and object files are in the same directory. Note that if source and object files are not in the same directory, the compiler will also have to be told where to put object files. You can do this using a %OD place marker (which expands to the directory specified using OBJDIR).

OBJEXT=

May be used to specify a non-standard object file extension. For example to specify that object files have the extension '.abc', specify

```
OBJEXT=ABC
```

This option may be useful for dealing with unusual compilers, but more commonly to allow AUTOMAKE to deal with processes other than compilation (for example, you could use AUTOMAKE to ensure that all altered source files are run through a pre-processor prior to compilation).

MODULE=

May be used to specify the name of the directory in which module files are stored.

```
MODULE=MODS\
```

The trailing '\' is optional. If `MODULE=` is not specified, AUTOMAKE assumes that source and module files are in the same directory. Note that if source and module files are not in the same directory, the compiler will also have to be told where to put module files. You can do this using a `%MO` place marker (which expands to the directory specified using `MODULE=`).

DEP=

May be used to over-ride the default dependency file name.

```
DEP=THISPROG.DEP
```

causes AUTOMAKE to store dependency data in 'thisprog.dep' instead of 'automake.dep'.

QUITONERROR

Specifies that AUTOMAKE should halt immediately if there is a compilation error.

NOQUITONERROR

Specifies that AUTOMAKE should not halt if there is a compilation error.

MAKEMAKE

Specifies that AUTOMAKE should create a text file called `AUTOMAKE.MAK` containing dependency information.

DEBUG

Causes AUTOMAKE to write debugging information to a file called `AUTOMAKE.DBG`.

LATESCAN

Delays scanning of source files until the last possible moment, and can, in some cases, remove the need for some scans. However this option is NOT compatible with Fortran 90 modules.

CHECK=

May be used to specify a command to be inserted after each compilation. A typical application would be to check for compilation errors. For example, under MS-DOS:

```
CHECK=IF ERRORLEVEL 2 GOTO QUIT
```

would cause the update procedure to abort if there is a compilation error.

Multi-Phase Compilation

Sometimes, more than one compilation phase is required. For example, if source files are stored in more than one directory, you will need a separate compilation phase for each directory. Multiple phases are also required if you have mixed C and Fortran source, or if you need special compilation options for particular source files.

The 'AND' keyword may be inserted in your configuration file to add a new compilation phase. You can reset the values of FILES=, COMPILE=, INCLUDE=, OBJDIR=, OBJEXT= and MODULE= for each phase. All default to the value used in the previous phase, except that OBJDIR= defaults to the new source directory.

The following example shows how this feature might be used with the LF95 compiler. The same principles apply to other compilers and other platforms.

```
# Example Configuration file for Multi-Phase
# Compilation
# Compilation 1 - files in current directory
LF95
INCLUDE=\include
FILES=*.f90
OBJDIR=obj
COMPILE=@lf95 -c %fi -i %id -o %od%sf%oe -tp -ol
AND
# Compilation 2 - files in utils\
# INCLUDE= defaults to previous value (\include)
# if OBJDIR= were not set, it would default to utils (NOT obj)
FILES=utils\*.f90
OBJDIR=utils\obj
COMPILE=@lf95 -c %fi -i %id -o %od%sf%oe -sav -chk
# Relink
TARGET=current.exe
LINK=@lf95 %ob -exe %ex
```

Automake Notes

- As AUTOMAKE executes, it issues brief messages to explain the reasons for all compilations. It also indicates when it is scanning through a file to look for INCLUDE statements.
- If for any reason the dependency file is deleted, AUTOMAKE will create a new one. Execution of the first AUTOMAKE will be slower than usual, because of the need to regenerate the dependency data.
- AUTOMAKE recognizes the INCLUDE statements in all common variants of Fortran and C, and can be used with both languages.
- When AUTOMAKE scans source code to see if it contains INCLUDE statements, it recognizes the following generalized format:
- Optional spaces at the beginning of the line followed by..
- An optional compiler control character, '%', '\$' or '#', followed by..

The word INCLUDE (case insensitive) followed by..

An optional colon followed by..

The file name, optionally enclosed between apostrophes, quotes or angled brackets. If the file name is enclosed in angled brackets, it is assumed to be in one of the directories specified using the `SYSINCLUDE` keyword. Otherwise, AUTOMAKE looks in the source file directory, and if it is not there, in the directories specified using the `INCLUDE` keyword.

- If AUTOMAKE cannot find an `INCLUDE` file, it reports the fact to the screen and ignores the dependency relationship.
- AUTOMAKE is invoked using a batch file called `AM.BAT`. There is seldom any reason to modify the command file, though it is very simple to do so if required. It consists of two (or three) operations:

Execute AUTOMAKE. AUTOMAKE determines what needs to be done in order to update the system, and writes a batch file to do it. The switches which may be appended to the AUTOMAKE command are:

`TO=` specifies the name of the output command file created by AUTOMAKE.

`FIG=` specifies the name of the AUTOMAKE configuration file.

Execute the command file created by AUTOMAKE.

Delete the command file created by AUTOMAKE. This step is, of course, optional.

8

The Sampler Tool

Tuning a program can significantly reduce its execution time. A specific section of a program may take most of the processing time, so tuning that section may greatly speed up the processing of the program. The Sampler tool helps you tune programs and detect bottlenecks.

Remarks:

1. When you tune a program, start by checking the cost for each function. If the cost of a function is high, the following two factors may be causes:
 - a. The function may include a redundant section.
 - b. The function itself may have no problems; however, it may be called excessively.

For the first cause, check the cost of the function source. For the second cause, check the source cost of the function calling this function.

2. The *cost* described in this chapter is the summed result of execution locations extracted per a specific time unit (per second, for example) based on a given sampling frequency. To illustrate, the cost of function *f* in a program is the number of locations belonging to *f* from the locations extracted per a specific time value.

Starting and Terminating the Sampler

Starting the Sampler

There are two ways to start the sampler:

1. From the Windows desktop icon
2. From the Windows command prompt

Starting from the Sampler icon

Start the sampler by double-clicking the Sampler icon on your Windows desktop if you chose to have an icon at installation time; otherwise start it from the Start | Programs menu.

Starting from the Command prompt

Type

SAMP

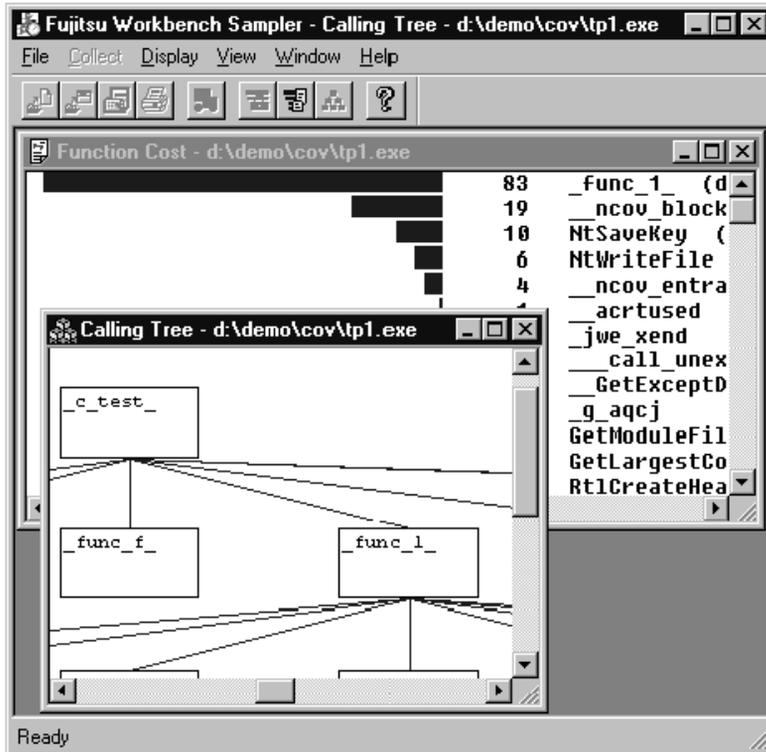
followed by the <Enter> key.

Terminating the Sampler

To terminate the sampler, choose the Exit Sampler command from the File menu in the sampler window.

The Sampler Window

See the figure below for the sampler window.



The above figure illustrates the following items:

1. The Toolbar, which contains icons that activate frequently used commands. These commands can also be executed from the Menu Bar.

2. The Status Bar, which displays an explanation of each menu command.
3. The “Function Cost” and “Calling Tree” windows, which are described later in this chapter.

Sampler Menus

The sampler menus are outlined below.

File Menu

The table below lists the File menu commands.

Table 18: Commands for the File Menu

Command	Function
Select Files	Open dialog for selecting executable (.exe) file and specifying sampling data output (.smp) file.
Print	Print the displayed tuning information
Print Preview	Display pages as they would appear if printed
Print Setup	Configure the printer
Exit Sampler	Terminate the Sampler

Remarks:

- The “Print” and “Print Preview” commands are displayed only if tuning information is being displayed.
- The “Print” and “Print Preview” commands cannot be used for the function calling relationship diagram.

Sampler Menu

The table below lists the Sampler menu commands.

Table 19: Commands in the Sampler Menu

Command	Function
Execute	Run the program to collect tuning information
Executing Options	Specify command line arguments and runtime options to the program and execute the program to collect tuning information
Function Cost	Display the cost for each function
Source Cost	Display the cost for each program unit
Calling Tree	Display the function calling relationship diagram (calling tree)
Program Type: Fortran	Specify whether to display Fortran program information for the Sampler Data
Program Type: C	Specify whether to display C program information for the Sampler Data
Program Type: Other	Specify whether to display information other than C or Fortran program information for the Sampler Data.
Source File Directory	Specify the source file directory

View Menu

The table below lists the View menu commands.

Table 20: Commands in the View Menu

Command	Function
Status Bar	Specify whether to display the status bar.
Toolbar	Specify whether to display the toolbar.
File Information	Specify whether to display file information

Window Menu

The table below lists the Window menu commands.

Table 21: Commands in the Window Menu

Command	Function
Cascade	Display all open windows so that they overlap, revealing the Title bar for each window.
Tile Horizontally	Display the listed tuning information from left to right.
Tile Vertically	Display the listed tuning information from top to bottom.
Arrange Icons	Arrange all the icons along the bottom of the window.
Close All	Close all open windows

Note: the Window menu is displayed only if tuning information is being displayed.

Help Menu

The table below lists the Help menu commands.

Table 22: Commands in the Help Menu

Command	Function
Help Topics	Display the Sampler help topics
About Sampler	Display version information for the Sampler.

Using the Sampler

The functions of the sampler are listed below. This section explains how to use these functions.

- Collecting the tuning information
- Displaying the tuning information

Collecting Tuning Information

In order to generate tuning information, the program must be compiled with the `-trace` option (see "`[/N]TRACE`" on page 33). To collect tuning information, run the program once, following the steps outlined below:

1. In the sampler, select the "Select Files..." command from the File menu. The Select Files dialog box appears.
2. Specify the Sample Data File, either by typing it in or browsing. The file must have an extension of `.smp`. Note that selection through browsing will set the default directory.
3. Specify the Executable File, either by typing it in or browsing. The file must have an extension of `.exe`.
4. Select one of the following methods of running the program:

To run the program with the existing execution options:

- a. Select the "Execute" command from the Sampler menu to run the program and collect its tuning information, allowing the program to terminate normally.
- b. To abort execution, click the Stop button in the window that is displayed while the program is running (this may interfere with generation of sampler data).

To run the program with modified execution options (i.e., command-line arguments):

- a. Select the "Executing Options" command from the Sampler menu. The Executing Options dialog box appears.
- b. In the Executing Options dialog box, specify the executing option. If the user program uses default input-output, specify a redirection function such as "`<<`" or "`>>`" in the executing option.
- c. Click the OK button.
- d. Execute the program and collect its tuning information, allowing it to terminate normally.
- e. To abort execution, click the Stop button in the window that is displayed while the program is running (this may interfere with generation of sampler data).

Note: In Windows 9x, if the message "Out of environment space" is displayed while running a console-mode program from the Sampler, it means the environment space of the DOS shell must be increased. This may be accomplished by adding the line (assuming that the system install directory is "`c:\windows`")

```
SHELL=C:\WINDOWS\COMMAND.COM /P /E:32768
```

to the `CONFIG.SYS` file. It may also be accomplished by modifying the "Initial Environment" property of the file `COMMAND.COM` using Windows explorer.

Displaying Tuning Information

The sampler displays the following three items of tuning information.

- Cost for each function
- Cost per line of the source level
- Function calling relationship diagram

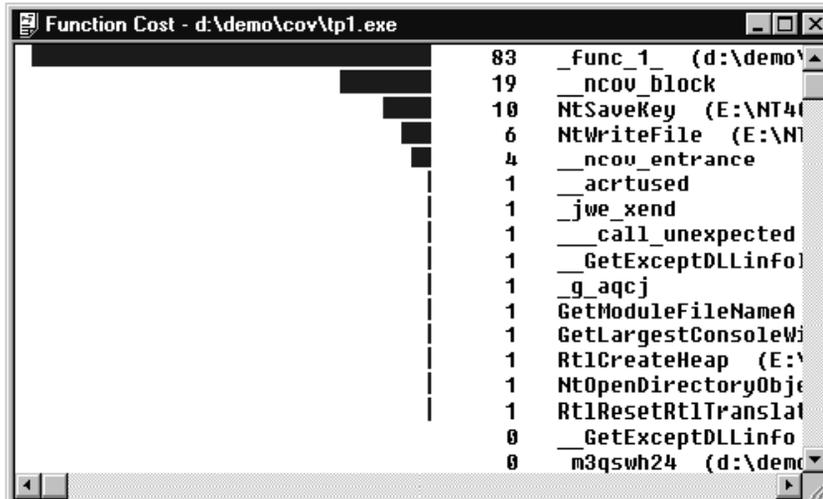
Recompilation is not required for an object generated with the Fujitsu Fortran compiler. To use the C compiler, specify the /Kline option at compilation.

The method of displaying each item of information is listed below.

Displaying the Cost for Each Function

Do the following to display the cost for each program unit:

In the sampler window, select the `Function` command from the `Display` menu. The function cost window appears. (See the figure below.)



Displaying the Cost Per Line

Do the following to display the cost per line of the source level:

1. In the sampler window, select the `Source` command from the `Display` menu. The `Open File` dialog box appears.
2. In the `Open File` dialog box, select the corresponding source program and click the `Open` button. The cost per line of the source level appears. (See the figure below for function cost per line of the source level window.)

The following procedures can also be used to display the cost per line of the source level:

Double clicking the function name that corresponds to the source code in the function cost window also shows the cost per line of the source code level.

```

214          ivcomp = 0
215          ivcorr = -110
216          do 888 kkk = 1,10000000
217      69          ivcomp = i2n016(1,1)*i2n016(2,3)
218      11 40160      if (ivcomp + 110) 20160, 10160,
219      36 10160      ivpass = ivpass + 1
220                  go to 0161
221          20160      ivfail = ivfail + 1
222      31 0161      continue
223                  ivtnum = 17
224      29          cvcomp = ' '
225                  ivcomp = 0
226      30          cvcorr = 'c001'
227      80          cvcomp = c2n001(0,1)
228     132 888      continue
229                  if (cvcomp .eq. 'c001') ivcomp
230                  if (ivcomp - 1) 20170, 10170, 2
231          10170      ivpass = ivpass + 1

```

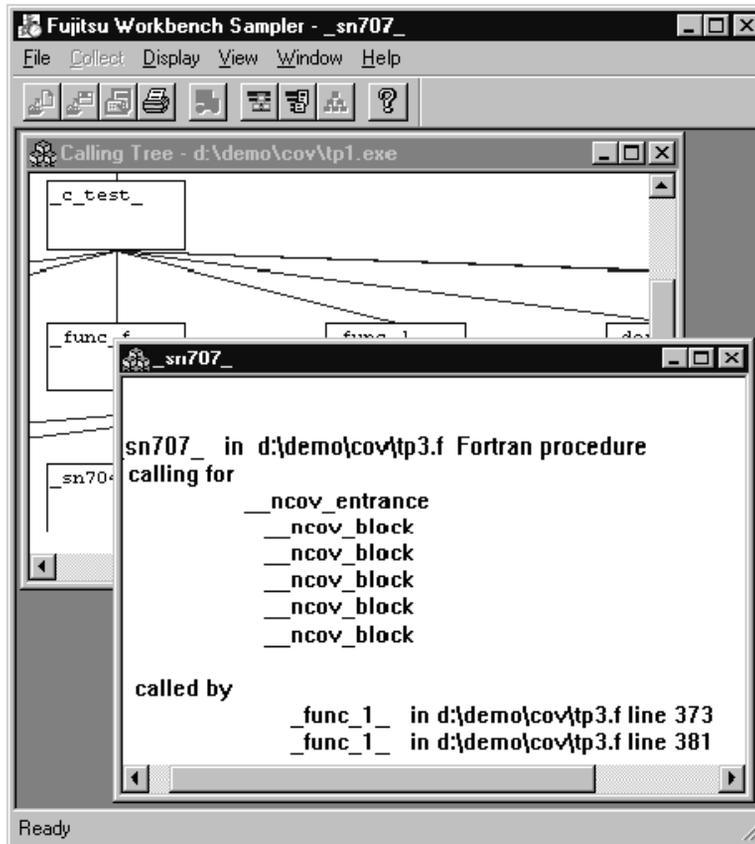
The Calling Relationship Diagram

Do the following to display the function calling relationship diagram.

1. In the sampler window, select the Reference command from the Display menu.
2. The calling tree window appears. (See the figure below for the calling tree window.)

Click the left button in the box of file names in the calling tree window to display the Focus and Detail menus. Select the Focus menu to display the calling relationship diagram from the function; select the Detail menu to display detailed information. If the Focus Level is more

than 1, press the left button outside the box to display the Top Level and Previous Level menus. Select the Top Level menu to display the relationship diagram of jump level 1. Select the Previous Level menu to display the previous relationship diagram.



Note: The cost information per line of the source level may differ slightly from the actual cost, because it is affected by the measuring machine status when information is collected (such as machine load status, number of logging users, and number of demons). The cost for each function always has about the same rate for the same program.

9

The Coverage Tool

One approach to program testing is to verify the operation range and coverage of the program execution. The Coverage Tool provides the following information for programs coded in the Fortran or C language:

- Executed and non-executed section information for each basic unit of execution flow
- Execution coverage information for each subroutine and function

Starting and Terminating the Coverage Tool

Starting the Coverage Tool

There are two ways to start the Coverage Tool:

1. From the Windows desktop icon
2. From the Windows command prompt

Starting from the desktop icon

Start by double-clicking the Coverage Tool icon on your Windows desktop if you chose to have an icon at installation time; otherwise start it from the `Start | Programs` menu.

Starting from the Command prompt

Type

```
COV
```

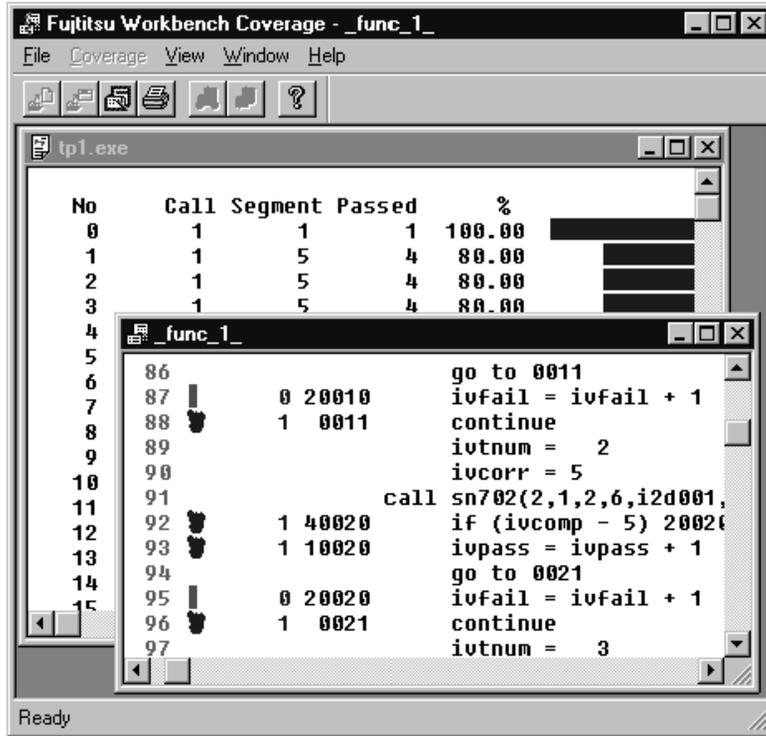
followed by the <Enter> key.

Terminating the Coverage Tool

In the coverage window, terminate the coverage tool by selecting the Exit Coverage command from the File menu.

Coverage Window

See the figure below for the coverage window.



The above figure illustrates the following items:

1. The Toolbar, which contains icons that activate frequently used commands. These commands can also be executed from the Menu Bar.
2. The Status Bar, which displays an explanation of each menu command.

Coverage Menus

The coverage menus are outlined below.

File Menu

The table below lists the File menu commands.

Table 23: Commands in the File Menu

Command	Function
Select Files	Open dialog for selecting executable (.exe) file and specifying coverage data output (.cov) file
Print	Print the coverage information being displayed.
Print Preview	Display image as it would be printed
Print Setup	Specify printer configuration.
File Search Path	Specify the path for searching for the file.
Exit Coverage	Terminate the coverage tool.

Note: the Print, and Print Preview commands are displayed only if coverage information is being displayed.

Coverage Menu

The table below lists the Coverage menu commands.

Table 24: Commands in the Coverage Menu

Command	Function
Execute	Run the program to collect coverage information.
Executing Options	Submit command line argument(s) and runtime options to the program and execute the program to collect coverage information
Execution Coverage Rate	Display the coverage information for all program units
Source File Directory	Specify the source file directory

View Menu

The table below lists the View menu commands.

Table 25: Commands in the View Menu

Command	Function
Status Bar	Specifies whether to display the status bar.
Toolbar	Specifies whether to display the toolbar.

Window Menu

The table below lists the Window menu commands.

Table 26: Commands in the Window Menu

Command	Function
Cascade	Displays all open windows so that they overlap, revealing the Title Bar for each window.
Tile Horizontally	Displays the listed coverage information from left to right.
Tile Vertically	Displays the listed coverage information from top to bottom.
Arrange Icons	Arranges all the icons along the bottom of the window.
Close All	Close all open windows

Note: the Window menu is displayed only if coverage information is being displayed.

Help Menu

The table below lists the Help menu commands.

Table 27: Commands in the Help Menu

Command	Function
Help	Displays the Coverage Tool help topics
About Coverage	Displays version information for the coverage tool.

Using the Coverage Tool

To compile a source program for the collection of coverage information, you must specify `-cover` as an option at compilation. If the source program is compiled without the `-cover` option specified, coverage information is not collected.

The coverage functions are listed below.

- Collecting coverage information
- Displaying coverage information

Collecting Coverage Information

To collect coverage information, run the program once.

Do the following to collect the information:

1. In the Coverage tool, select the “Select Files...” command from the File menu. The Select Files dialog box appears.
2. Specify the Coverage Data File, either by typing it in or browsing. The file must have an extension of `.cov`. Note that selection through browsing will set the default directory.
3. Specify the Executable File, either by typing it in or browsing. The file must have an extension of `.exe`.
4. Select one of the following methods for executing the program:

To execute the program with the existing execution options:

- a. Select the “Execute” command from the Coverage menu to run the program and collect its coverage information, allowing the program to terminate normally.
- b. To abort execution, click the Stop button in the window that is displayed while the program is running (this may interfere with generation of coverage data).

To execute the program with modified execution options (i.e., command-line arguments):

- a. Select the “Executing Options” command from the Coverage menu. The Executing Options dialog box appears.
- b. In the Executing Options dialog box, specify the executing option. If the user program uses default input-output, specify a redirection function such as `'<'` or `'>'` in the option.
- c. Click the OK button.
- d. Run the program and collect its coverage information, allowing the program to terminate normally.
- e. To abort execution, click the Stop button in the window that is displayed while the program is executing (this may interfere with generation of coverage data).

Note: In Windows 9x, if the message “Out of environment space” is displayed while running a console-mode program from the Coverage Tool, it means the environment space of the DOS shell must be increased. This may be accomplished by adding the line (assuming that the system install directory is “c:\windows”)

```
SHELL=C:\WINDOWS\COMMAND.COM /P /E:32768
```

to the CONFIG.SYS file. It may also be accomplished by modifying the “Initial Environment” property of the file COMMAND.COM using Windows explorer.

Storing & Merging Coverage Information

Collected coverage information can be stored. You can update and display the stored information by assigning another argument to the executable program (merging the information). If the program being tested requires input data, you can put sample data into a file and then use that data for testing the program.

Storing Coverage Information

Coverage information is stored in the file specified by the “Select Coverage Data File” command in the File menu.

Merging Coverage Information

Follow these steps to merge coverage information with the existing coverage data file:

1. Use the “Select Coverage Data File” command in the File menu to specify the data file containing the collected coverage information.
2. Use the “Select Executable File” command in the File menu to specify the same executable file generated by (1) above as the Executable File.
3. Select the “Execute” or “Executing Options” command from the Coverage menu to execute the program and collect its coverage information.

The new coverage information is now stored in the specified data file.

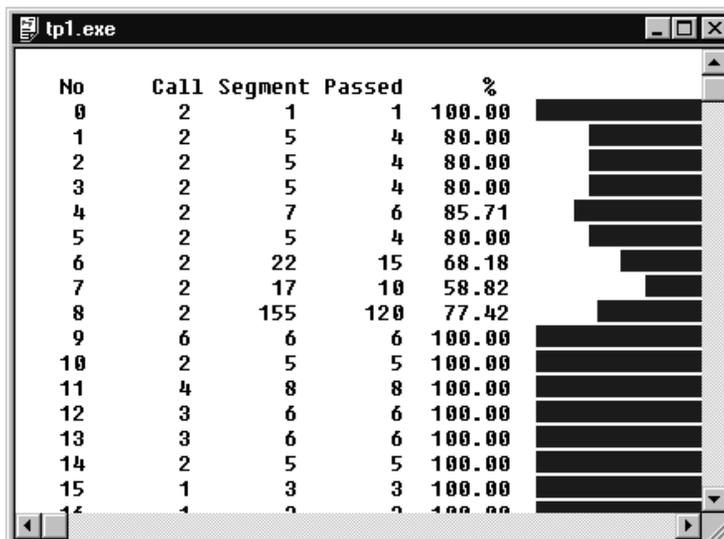
Displaying Coverage Information

The following items are displayed in the coverage information:

- Run coverage rate for each subroutine and function
- Executed and non-executed section corresponding to the source code

Follow these steps to display the information:

1. Select the “Select Coverage Data File” command from the File menu. The Coverage Data File dialog box appears.
2. In the Coverage Data File dialog box, specify the coverage data file.
3. Select the “Select Executable File” command from the File menu. The Executable File dialog box appears.
4. In the Executable File dialog box, specify the executable file.
5. Select the “Function” command from the Display menu to display the execution coverage rate for each subroutine and function. (See the first figure below.) Then double-click the subroutine or function to display the executed and non-executed sections of the corresponding source code. (See the second figure below.)



```

84  1 40010  if (ivcomp + 47) 20010, 10010,
85  1 10010  ivpass = ivpass + 1
86  1 10010  go to 0011
87  0 20010  ivfail = ivfail + 1
88  1 0011   continue
89  1 0011   ivtnum = 2
90  1 0011   ivcorr = 5
91  1 0011   call sn702(2,1,2,6,i2d001,i2d002,i2d
92  1 40020  if (ivcomp - 5) 20020, 10020, 2
93  1 10020  ivpass = ivpass + 1
94  1 10020  go to 0021
95  0 20020  ivfail = ivfail + 1
96  1 0021   continue
97  1 0021   ivtnum = 3
98  1 0021   ivcorr = 17
99  1 0021   call sn702(3,1,2,6,i2d001,i2d002,i2d
100 1 40030  if (ivcomp - 17) 20030, 10030,
101 1 10030  ivpass = ivpass + 1
    
```

Remarks:

1. The coverage tool shows the executed and non-executed parts of each block. A block is a set of statements that do not change the control of execution. The following instances may separate a block:

- a. Fortran: IF, CASE, DO and GOTO
- b. C language: if, case, for and label

2. The following marks are placed at the beginning of the block:

- a. Blue foot mark: the block which is executed for the first time
- b. White foot mark: the block which is executed for the second or subsequent times.
- c. Stick mark: the block that is never executed.

10

Utility Programs

This chapter documents the following utility programs:

- CFG386.EXE
- HDRSTRIP.F90
- SEQUNF.F90
- TRYBLK.F90
- UNFSEQ.EXE
- WHICH.EXE
- RSE.EXE

CFG386.EXE

CFG386.EXE allows you configure switches directly into the linker to change its default operation. These switches are automatically processed every time the program is run.

If CFG386 is run with the name of a linker executable file as the only argument on the command line, it displays the current contents of the linker's configuration block. If it is run without any arguments, a list of valid CFG386 command-line switches is displayed.

Configuring New Switches

Following the linker name, you list one or more command switches. The switches are given in the same format they are given on the command line for the program being configured. The specified switches are added to the configuration block of the program after any switches that are already there from any previous configurations.

Example

```
cfg386 386link -nomap -libpath c:\lf9550\lib
```

CFIG386 does not check the values of any of the switches or switch parameters it stores in the program's configuration block. Thus, it is possible to configure invalid switch values into the linker. You should always run the program after configuring it to make sure that the configured switch values have the desired effect.

-Clear

The `-clear` switch causes CFIG386 to erase the current contents of the program's configuration block. Any switches specified after the clear switch are added to the just-cleared configuration block.

Example

```
cfig386 -clear 386link
cfig386 -clear 386link -nomap
```

HDRSTRIP.F90

`HDRSTRIP.F90` is a Fortran source file that you can compile, link, and execute with LF95. It converts LF90 direct-access files to LF95 style.

PENTEST.F90

`PENTEST.F90` is a Fortran source file that you can compile, link, and execute with LF95. It tests for the Pentium flaw that affects certain floating-point operations. It will notify you if your chip exhibits the flaw.

SEQUNF.F90

`SEQUNF.F90` is a Fortran source file that you can compile, link, and execute with LF95. It converts LF90 unformatted sequential files to LF95 style.

TRYBLK.F90

`TRYBLK.F90` is a Fortran source file you can build with LF95. It tries a range of `BLOCK-SIZES` and displays an elapsed time for each. You can use the results to determine an optimum value for your PC to specify in your programs. Note that a particular `BLOCKSIZE` may not perform as well on other PC's.

UNFSEQ.EXE

UNFSEQ.EXE is an executable file that converts LF95 unformatted sequential files to LF90 style.

WHICH.EXE

WHICH.EXE is a utility to search the current directory and directories on the DOS path for all .COM, .EXE, and .BAT files matching a filename specification. It is useful for determining which of a number of programs or batch files with the same name will be invoked.

Syntax

```
which filename
```

where *filename* is the name of a .COM, .EXE, or .BAT file without the extension.

Example

If the command

```
which make
```

displays the following

```
C:\LF9550\BIN\MAKE.EXE      47541  17:05  07-07-1994
C:\F77L3\BIN\MAKE.EXE      46237  13:16  05-22-1994
```

you would know that the MAKE.EXE program in the c:\lf9550\bin directory would be invoked because c:\lf9550\bin is listed before c:\f7713\bin on the DOS path.

RSE.EXE

RSE redirects to standard output all output which would normally go to standard error.

Syntax

```
rse exename
```

where *exename* is the name of an executable file. RSE is particularly useful when attempting to redirect runtime error messages to a disk file. Normally, runtime error messages from executables created by LF95 will be sent to standard error. If you try to redirect program output as follows

```
myprog > diskfile
```

program output will go to the file, but runtime error messages will go only to the screen. To redirect runtime error messages to the file, use

```
rse myprog > diskfile
```


11

Programming Hints

This appendix contains information that may help you create better LF95 programs.

Efficiency Considerations

In the majority of cases, the most efficient solution to a programming problem is one that is straightforward and natural. It is seldom worth sacrificing clarity or elegance to make a program more efficient.

The following observations, which may not apply to other implementations, should be considered in cases where program efficiency is critical:

- Start each array dimension at zero (not at one, which is the default). Thus, declare an array *A* to be `A(0:99)`, not `A(100)`.
- One-dimensional arrays are more efficient than two, two are more efficient than three, etc.
- Make a direct file record length a power of two.
- Unformatted input/output is faster for numbers.
- Formatted CHARACTER input/output is faster using:

```
CHARACTER*256 C
```

than:

```
CHARACTER*1 C(256)
```

Side Effects

LF95 arguments are passed to subprograms by address, and the subprograms reference those arguments as they are defined in the called subprogram. Because of the way arguments are passed, the following side effects can result:

- Declaring a dummy argument as a different numeric data type than in the calling program unit can cause unpredictable results and NDP error aborts.
- Declaring a dummy argument to be larger in the called program unit than in the calling program unit can result in other variables and program code being modified and unpredictable behavior.
- If a variable appears twice as an argument in a single CALL statement, then the corresponding dummy arguments in the subprogram will refer to the same location. Whenever one of those dummy arguments is modified, so is the other.
- Function arguments are passed in the same manner as subroutine arguments, so that modifying any dummy argument in a function will also modify the corresponding argument in the function invocation:

$$y = x + f(x)$$

The result of the preceding statement is undefined if the function f modifies the dummy argument x .

File Formats

Formatted Sequential File Format

Files controlled by formatted sequential input/output statements have an undefined length record format. One Fortran record corresponds to one logical record. The length of the undefined length record depends on the Fortran record to be processed. The max length may be assigned in the OPEN statement RECL= specifier. The carriage-return/line-feed sequence terminates the logical record. If the \$ edit descriptor or \ edit descriptor is specified for the format of the formatted sequential output statement, the Fortran record does not include the carriage-return/line-feed sequence.

Unformatted Sequential File Format

Files processed using unformatted sequential input/output statements have a variable length record format. One Fortran record corresponds to one logical record. The length of the variable length record depends on the length of the Fortran record. The length of the Fortran record includes 4 bytes added to the beginning and end of the logical record. The max length may be assigned in the OPEN statement RECL= specifier. The beginning area is used when an unformatted sequential statement is executed. The end area is used when a BACKSPACE statement is executed.

Direct File Format

Files processed by unformatted direct input/output statements have a fixed length record format. One Fortran record can correspond to more than one logical record. The record length must be assigned in the OPEN statement RECL= specifier. If the Fortran record terminates within a logical record, the remaining part is padded with binary zeros. If the length of the Fortran record exceeds the logical record, the remaining data goes into the next record.

Transparent File Format

Files opened with ACCESS="TRANSPARENT" or FORM="BINARY" are processed as a stream of bytes with no record separators. While any format of file can be processed transparently, you must know its format to process it correctly.

Determine Load Image Size

To determine the load image size of a protected-mode program, add the starting address of the last public symbol in the linker map file to the length of that public symbol to get an approximate load image memory requirement (not execution memory requirement).

Link Time

Due to the error checking that 386LINK does, certain code can cause the linker to take longer. For example, using hundreds to thousands of named COMMON blocks causes the linker to slow down. Most of the additional time is spent in processing the names themselves because Windows (requires certain ordering rules to be followed within the executable itself).

You can reduce the link time by reducing the number of named COMMON blocks you use. Instead of coding:

```
common /a1/ i
common /a2/ j
common /a3/ k
...
common /a1000/ k1000
```

code:

```
common /a/ i,j,k, ..., k1000
```

Link time may also be reduced by using the -NOMAP switch.

Year 2000 compliance

The "Year 2000" problem arises when a computer program uses only two digits to represent the current year and assumes that the current century is 1900. A compiler can look for indications that this might be occurring in a program and issue a warning, but it cannot foresee every occurrence of this problem. It is ultimately the responsibility of the programmer to correct the situation by modifying the program. The most likely source of problems for Fortran programs is the use of the obsolete DATE() subroutine. Even though LF95 will compile and link programs that use DATE(), its use is strongly discouraged; the use of DATE_AND_TIME(), which returns a four digit date, is recommended in its place.

LF95 can be made to issue a warning at runtime whenever a call to DATE() is made. This can be accomplished by running a program with the runtime options `-w1,ry,li` for example,

```
myprog.exe -w1,ry,li
```

For more information on runtime options, see "*Runtime Options*" on page 143.

Limits of Operation.

Table 28: LF95 Limits of Operation

Item	Maximum
program size	4 Gigabytes or available memory (including virtual memory), whichever is smaller
number of files open concurrently	250, including pre-connected units 0, 5, and 6
Length of CHARACTER datum	65,000 bytes
I/O block size	65,000 bytes
I/O record length	2,147,483,647 bytes
I/O file size	2,147,483,647 bytes
I/O maximum number of records	2,147,483,647 divided by the value of RECL= specifier
nesting depth of function, array section, array element, and substring references	255
nesting depth of DO, CASE, and IF statements	50
nesting depth of implied-DO loops	25
nesting depth of INCLUDE files	16

Table 28: LF95 Limits of Operation

Item	Maximum
number of array dimensions	7
array size	<p>The compiler calculates T for each array declaration to reduce the number of calculations needed for array sections or array element addresses. The absolute value of T obtained by the formula below must not exceed 2147483647, and the absolute value must not exceed 2147483647 for any intermediate calculations:</p> $T = l1 \times s + \sum_{i=2}^n \left\{ li \times \left(\prod_{m=2}^i dm \ominus 1 \times s \right) \right\}$ <p> n: Array dimension number s: Array element length l: Lower bound of each dimension d: Size of each dimension T: Value calculated for the array declaration </p>

12

Runtime Options

The behavior of the LF95 runtime library can be modified at the time of execution by a set of commands which are submitted via the command line when invoking the executable program, or via shell environment variables. These runtime options can modify behavior of input/output operations, diagnostic reporting, and floating-point operations.

Runtime options submitted on the command line are distinguished from user-defined command line arguments by using a character sequence that uniquely identifies the runtime options, so that they will not interfere with the passing of regular command line arguments that the user's program might be expecting to obtain via the GETCL(), GETPARAM(), or GETARG() functions.

Command Format

Runtime options and user-defined executable program options may be specified as command option arguments of an execution command. The runtime options use functions supported by the LF95 runtime library. Please note that these options are *case-sensitive*.

The format of runtime options is as follows:

exe_file [/Wl,[runtime options],...] [*user-defined program arguments*]...

Where *exe_file* indicates the user's executable program file. The string "/Wl," (or "-Wl,") must precede any runtime options, so they may be identified as such and distinguished from user-defined program arguments. Note that it is W followed by a lowercase L (not the number one). Please note also that if an option is specified more than once with different arguments, the last occurrence is used.

Command Shell Variable

As an alternative to the command line, the shell variable FORT90L may be used to specify runtime options. Any runtime options specified in the command line are combined with those specified in FORT90L. The command line arguments take precedence over the corresponding options specified in the shell variable FORT90L.

The following examples show how to use the shell variable FORT90L (the actual meaning of each runtime option will be described in the sections below):

Example 1:

Setting the value of shell variable FORT90L and executing the program as such:

```
set FORT90L=-Wl,e99,le  
a.exe -Wl,m99 /k
```

has the same effect as the command line

```
a.exe -Wl,e99,le,m99 /k
```

The result is that when executing the program a.exe, the runtime options e99, le, and m99, and user-defined executable program argument /k are in effect.

Example 2:

When the following command lines are used,

```
set FORT90L=-Wl,e10  
a.exe -Wl,e99
```

the result is that a.exe is executed with runtime option /e99 is in effect, overriding the option e10 set by shell variable FORT90L.

Execution Return Values

The following table lists possible values returned to the operating system by an LF95 executable program upon termination and exit. These correspond to the levels of diagnostic output that may be set by various runtime options:

Table 29: Execution Return Values

Return value	Status
0	No error or level I (information message)
4	Level W error (warning)
8	Level E error (medium)
12	Level S error (serious)
16	Limit exceeded for level W, E, S error, or a level U error (Unrecoverable) was detected
240	Abnormal termination
Other	Forcible termination

Standard Input and Output

The default unit numbers for standard input, output, and error output for LF95 executable programs are as follows, and may be changed to different unit numbers by the appropriate runtime options:

Standard input: Unit number 5

Standard output: Unit number 6

Standard error output: Unit number 0

Runtime Options

Runtime options may be specified as arguments on the command line, or in the FORT90L shell variable. This section explains the format and functions of the runtime options. Please note that all runtime options are *case-sensitive*.

The runtime option format is as follows:

```
/Wl[,Cunit][,M][,Q][,Re][,Rm:file][,Tunit][,a][,dnum][,enum][,gnum][,i][,lelvl][,munit][,n][,punit][,q][,runit][,u][,x]
```

When runtime options are specified, the string “/Wl” (where l is lowercase L) is required at the beginning of the options list, and the options must be separated by commas. If the same runtime option is specified more than once with different arguments, the last occurrence is used.

Example:

```
a.exe /Wl,a,p10,x
```

Description of Options

C or C[unit]

The C option specifies how to process an unformatted file of IBM370-format floating-point data using an unformatted input/output statement. When the C option is specified, the data of an unformatted file associated with the specified unit number is regarded as IBM370-format floating-point data in an unformatted input/output statement. The optional argument *unit* specifies an integer from 0 to 2147483647 as the unit number. If optional argument *unit* is omitted, the C option is valid for all unit numbers connected to unformatted files. When the specified unit number is connected to a formatted file, the option is ignored for the file. When the C option is not specified, the data of an unformatted file associated with unit number *unit* is regarded as IEEE-format floating-point data in an unformatted input-output statement.

Example:

```
a.exe /Wl,C10
```

M

The M option specifies whether to output the diagnostic message (jwe0147i-w) when bits of the mantissa are lost during conversion of IBM370-IEEE-format floating-point data. If the M option is specified, a diagnostic message is output if conversion of IBM370-IEEE-format floating-point data results in a bits of the mantissa being lost. When the M option is omitted, the diagnostic message (jwe0147i-w) is not output.

Example:

```
a.exe /Wl,M
```

Q

The Q option suppresses padding of an input field with blanks when a formatted input statement is used to read a Fortran record. This option applies to cases where the field width needed in a formatted input statement is longer than the length of the Fortran record and the file was not opened with an OPEN statement. The result is the same as if the PAD= specifier in an OPEN statement is set to NO. If the Q option is omitted, the input record is padded with blanks. The result is the same as when the PAD= specifier in an OPEN statement is set to YES or when the PAD= specifier is omitted.

Example:

```
a.exe /W1,Q
```

Re

Disables the runtime error handler. Traceback, error summaries, user control of errors by ERRSET and ERRSAV, and execution of user code for error correction are suppressed. The standard correction is processed if an error occurs.

Example:

```
a.exe /W1,Re
```

Ri

Disables runtime processing of quad precision exceptions.

Example:

```
a.exe /W1,Ri
```

Rm: *filename*

The Rm option saves the following output items to the file specified by the *filename* argument:

- Messages issued by PAUSE or STOP statements
- Runtime library diagnostic messages
- Traceback map
- Error summary

Example:

```
a.exe /W1,Rm:errors.txt
```

Ry

Enforces Y2K compliance at runtime by generating an i-level (information) diagnostic whenever code is encountered which may cause problems after the year 2000A.D. Must be used in conjunction with li option in order to view diagnostic output.

Example:

```
a.exe /W1,Ry,li
```

T or T[*u_no*]

Big endian integer data, logical data, and IEEE floating-point data is transferred in an unformatted input/output statement. The optional argument *u_no* is a unit number, valued between 0 and 2147483647, connected with an unformatted file. If *u_no* is omitted, T takes effect for all unit numbers. If both T and T*u_no* are specified, then T takes effect for all unit numbers.

Example:

```
a.exe /WL,T10
```

a

When the a option is specified, an abend is executed forcibly following normal program termination. This processing is executed immediately before closing external files.

Example:

```
a.exe /WL,a
```

d[num] 1

The d option determines the size of the input/output work area used by a direct access input/output statement. The d option improves input/output performance when data is read from or written to files a record at a time in sequential record-number order. If the d option is specified, the input/output work area size is used for all units used during execution.

To specify the size of the input/output work area for individual units, specify the number of Fortran records in the shell variable `FUnnBF` where `nn` is the unit number (see “*Shell Variables for Input/Output*” on page 151 for details). When the d option and shell variable are specified at the same time, the d option takes precedence. The optional argument `num` specifies the number of Fortran records, in fixed-block format, included in one block. The optional argument `num` must be an integer from 1 to 32767. To obtain the input/output work area size, multiply `num` by the value specified in the `RECL=` specifier of the `OPEN` statement. If the files are shared by several processes, the number of Fortran records per block must be 1. If the d option is omitted, the size of the input/output work area is 4K bytes.

Example:

```
a.exe /WL,d10
```

e[num]

The e option controls termination based on the total number of execution errors. The option argument `num`, specifies the error limit as an integer from 0 to 32767. When `num` is greater than or equal to 1, execution terminates when the total number of errors reaches the limit. If `enum` is omitted or `num` is zero, execution is not terminated based on the error limit. However, program execution still terminates if the Fortran system error limit is reached.

Example:

```
a.exe /WL,e10
```

gnum

The g option sets the size of the input/output work area used by a sequential access input/output statement. This size is set in units of kilobytes for all unit numbers used during execution. The argument `num` specifies an integer with a value of 1 or more. If the g option is omitted, the size of the input/output work area defaults to 8 kilobytes.

The `g` option improves input/output performance when a large amount of data is read from or written to files by an unformatted sequential access input/output statement. The argument `num` is used as the size of the input/output work area for all units. To avoid using excessive memory, specify the size of the input/output work area for individual units by specifying the size in the shell variable `fuxxbf`, where `xx` is the unit number (see “*Shell Variables for Input/Output*” on page 151 for details). When the `g` option is specified at the same time as the shell variable `fuxxbf`, the `g` option has precedence.

Example:

```
a.exe /Wl,g10
```

i

The `i` option controls processing of runtime interrupts. When the `i` option is specified, the Fortran library is not used to process interrupts. When the `i` option is not specified, the Fortran library is used to process interrupts. These interrupts are exponent overflow, exponent underflow, division check, and integer overflow. If runtime option `-i` is specified, no exception handling is taken. The `u` option must not be combined with the `i` option. Note that the `i` option does not control quad-precision exceptions (see “*Ri*” on page 147).

Example:

```
a.exe /Wl,i
```

lerrlvl errlvl: { i | w | e | s }

The `l` option (lowercase `L`) controls the output of diagnostic messages during execution. The optional argument `errlvl`, specifies the lowest error level, `i` (informational), `w` (warning), `e` (medium), or `s` (serious), for which diagnostic messages are to be output. If the `l` option is not specified, diagnostic messages are output for error levels `w`, `e`, and `s`. However, messages beyond the print limit are not printed.

i

The `li` option outputs diagnostic messages for all error levels.

w

The `lw` option outputs diagnostic messages for error levels `w`, `e`, `s`, and `u`.

e

The `le` option outputs diagnostic messages for error levels `e`, `s`, and `u`.

s

The `ls` option outputs diagnostic messages for error levels `s` and `u`.

Example:

```
a.exe /Wl,le
```

mu_no

The m option connects the specified unit number *u_no* to the standard error output file where diagnostic messages are to be written. Argument *u_no* is an integer from 0 to 2147483647. If the m option is omitted, unit number 0, the system default, is connected to the standard error output file. See *"Shell Variables for Input/Output"* on page 151 for further details.

Example:

```
a.exe /Wl,m10
```

n

The n option controls whether prompt messages are sent to standard input. When the n option is specified, prompt messages are output when data is to be entered from standard input using formatted sequential READ statements, including list-directed and namelist statements. If the n option is omitted, prompt messages are not generated when data is to be entered from standard input using a formatted sequential READ statement.

Example:

```
a.exe /Wl,n
```

pu_no

The p option connects the unit number *u_no* to the standard output file, where *u_no* is an integer ranging from 0 to 2147483647. If the p option is omitted, unit number 6, the system default, is connected to the standard output file. See *"Shell Variables for Input/Output"* on page 151 for further details.

Example:

```
a.exe /Wl,p10
```

q

The q option specifies whether to capitalize the E, EN, ES, D, Q, G, L, and Z edit output characters produced by formatted output statements. This option also specifies whether to capitalize the alphabetic characters in the character constants used by the inquiry specifier (excluding the NAME specifier) in the INQUIRE statement. If the q option is specified, the characters appear in uppercase letters. If the q option is omitted, the characters appear in lowercase letters. If compiler option `-neix` is in effect, the characters appear in uppercase letters so the q option is not required.

Example:

```
a.exe /W1,q
```

ru_no

The *r* option connects the unit number *u_no* to the standard input file during execution, where *u_no* is an integer ranging from 0 to 2147483647. If the *r* option is omitted, unit number 5, the system default, is connected to the standard input file. See "Shell Variables for Input/Output" on page 151 for further details.

Example:

```
a.exe /W1,r10
```

u

The *u* option controls floating point underflow interrupt processing. If the *u* option is specified, the system performs floating point underflow interrupt processing. The system may output diagnostic message `jwe0012i-e` during execution. If the *u* option is omitted, the system ignores floating point underflow interrupts and continues processing. The *i* option must not be combined with the *u* option.

Example:

```
a.exe /W1,u
```

x

The *x* option determines whether blanks in numeric edited input data are ignored or treated as zeros. If the *x* option is specified, blanks are changed to zeros during numeric editing with formatted sequential input statements for which no OPEN statement has been executed. The result is the same as when the BLANK= specifier in an OPEN statement is set to zero. If the *x* option is omitted, blanks in the input field are treated as null and ignored. The result is the same as if the BLANK= specifier in an OPEN statement is set to NULL or if the BLANK= specifier is omitted.

Example:

```
a.exe /W1,x
```

Shell Variables for Input/Output

This section describes shell variables that control file input/output operations

FUnn = filename

The *FUnn* shell variable connects units and files. The value *nn* is a unit number. The value *filename* is a file to be connected to unit number *nn*. The standard input and output files (FU05 and FU06) and error file (FU00) must not be specified.

The following example shows how to connect myfile.dat to unit number 10 prior to the start of execution.

Example:

```
set FU10=myfile.dat
```

FU n BF = size

The FU n BF shell variable specifies the size of the input/output work area used by a sequential or direct access input/output statement. The value nm in the FU n BF shell variable specifies the unit number. The size argument used for sequential access input/output statements is in kilobytes; the *size* argument used for direct access input/output statements is in records. The *size* argument must be an integer with a value of 1 or more. A *size* argument must be specified for every unit number.

If this shell variable and the *g* option are omitted, the input/output work area size used by sequential access input/output statements defaults to 1 kilobytes. The *size* argument for direct access input/output statements is the number of Fortran records per block in fixed-block format. The *size* argument must be an integer from 1 to 32767 that indicates the number of Fortran records per block. If this shell variable and the *d* option are omitted, the area size is 1K bytes.

Example 1:

Sequential Access Input/Output Statements.

When sequential access input/output statements are executed for unit number 10, the statements use an input/output work area of 64 kilobytes.

```
set FU10BF=64
```

Example 2:

Direct Access Input/Output Statements.

When direct access input/output statements are executed for unit number 10, the number of Fortran records included in one block is 50. The input/output work area size is obtained by multiplying 50 by the value specified in the RECL= specifier of the OPEN statement.

```
set FU10BF=50
```



Lahey Technical Support

Lahey Computer Systems takes pride in the relationships we have with our customers. We maintain these relationships by providing quality technical support, an electronic mail (e-mail) system, a web site, newsletters, product brochures, and new release announcements. The World Wide Web site has product patch files, new Lahey product announcements, lists of Lahey-compatible software vendors and information about downloading other Fortran related software. In addition, we listen carefully to your comments and suggestions.

Hours

Lahey's business hours are

7:45 A.M. to 5:00 P.M. Pacific Time Monday - Thursday

7:45 A.M. to 1:00 P.M. Pacific Time Friday

Telephone technical support is available

8:30 A.M. to 3:30 P.M. Pacific Time Monday - Thursday

8:30 A.M. to 12:00 P.M. Pacific Time Friday

We have several ways for you to communicate with us:

- TEL: (775) 831-2500 (PRO version only)
- FAX: (775) 831-8123
- E-MAIL: support@lahey.com
- ADDRESS: 865 Tahoe Blvd.
P.O. Box 6091
Incline Village, NV 89450-6091 U.S.A.

Technical Support Services

Lahey provides free technical support to registered users. This support includes assistance in the use of our software and in getting any bugs you may find in our software fixed. It does not include tutoring in how to program in FORTRAN or how to use any host operating system.

How Lahey Fixes Bugs

Lahey's technical support goal is to make sure you can create working executables using LF95. Towards this end, Lahey maintains a bug reporting and prioritized resolution system. We give a bug a priority based on its severity.

The definition of any bug's severity is determined by whether or not it directly affects your ability to build and execute a program. If a bug keeps you from being able to build or execute your program, it receives the highest priority. If you report a bug that does not keep you from creating a working program, it receives a lower priority. Also, if Lahey can provide a "workaround" to the bug, it receives a lower priority.

In recognizing that problems sometimes occur in changing software versions, Lahey allows you to revert to an earlier version of the software until Lahey resolves the problem.

Lahey continues to fix bugs in a numbered version of LF95 until 60 days after the next numbered version is released.

Contacting Lahey

To expedite support services, we prefer written or electronic communications via FAX or e-mail. These systems receive higher priority service and minimize the chances for any mistakes in our communications.

Before contacting Lahey Technical Support, we suggest you do the following to help us process your report.

- Determine if the problem is specific to code you created. Can you reproduce it using the demo programs we provide?
- If you have another machine available, does the problem occur on it?

Information You Provide

When contacting Lahey, please include or have available the information listed below.

For All Problems

1. The Lahey product name, serial, and version numbers.
2. A description of the problem to help us duplicate it. Include the exact error message numbers and/or message text.

For Compiler Problems

1. An example of the code that causes the problem. Please make the example as small as possible to shorten our response time and reduce the chances for any misunderstandings.
2. A copy of the LF95.FIG file (driver configuration file).
3. Command-line syntax and any options used for the driver or other tools.

For Other Problems

1. The brand and model of your system.
2. The type and speed of your CPU.

Lahey will respond promptly after we receive your report with either the solution to the problem or a schedule for solving the problem.

Technical Support Questionnaire

The Lahey Tech Support Questionnaire utility can be used to facilitate the gathering of critical information for your support request. It may even help you solve your problem on the spot. It presents a series of dialogs that will guide you to provide the most pertinent information, generating a file that you can attach to your e-mail to support@lahey.com. Start it from the LF95 toolbar in Lahey ED Developer, or from the Lahey/Fujitsu Fortran 95 folder in your Programs menu, or run TSQ.

World Wide Web Site

Our URL is <http://www.lahey.com>. Visit our web site to get the latest information and product patch and fix files and to access other sites of interest to Fortran programmers.

Lahey Warranties

Lahey's 30 Day Money Back Guarantee

Lahey agrees to unconditionally refund to the purchaser the entire purchase price of the product (including shipping charges up to a maximum of \$10.00) within 30 days of the original purchase date.

All refunds require a Lahey Returned Materials Authorization (RMA) number. Lahey must receive the returned product within 15 days of assigning you an RMA number. If you purchased your Lahey product through a software dealer, the return must be negotiated through that dealer.

Lahey's Extended Warranty

Lahey agrees to refund to the purchaser the entire purchase price of the product (excluding shipping) at any time subject to the conditions stated below.

All refunds require a Lahey Returned Materials Authorization (RMA) number. Lahey must receive the returned product in good condition within 15 days of assigning you an RMA number.

You may return a LF95 Language System if:

- It is determined not to be a full implementation of the Fortran 90 Standard and Lahey does not fix the deviation from the standard within 60 days of your report.
- Lahey fails to fix a bug with the highest priority within 60 days of verifying your report.
- All returns following the original 30 days of ownership are subject to Lahey's discretion. If Lahey has provided you with a source code workaround, a compiler patch, a new library, or a reassembled compiler within 60 days of verifying your bug report, the problem is considered by Lahey to be solved and no product return and refund is considered justified.

Return Procedure

You must report the reason for the refund request to a Lahey Solutions Representative and receive an RMA number. This RMA number must be clearly visible on the outside of the return shipping carton. Lahey must receive the returned product within 15 days of assigning you an RMA number. You must destroy the following files before returning the product for a refund:

- All copies of Lahey files delivered to you on the software disks and all backup copies.
- All files created by this Lahey Language System.

A signed statement of compliance to the conditions listed above must be included with the returned software. Copy the following example for this statement of compliance:

I, _____(your name), in accordance with the terms specified here, acknowledge that I have destroyed all backup copies of and all other files created with the Lahey software. I no longer have in my possession any copies of the returned files or documentation. Any violation of this agreement will bring legal action governed by the laws of the State of Nevada.

Signature:

Print Name:

Company Name:

Address:

Telephone:

Product: Version: Serial #:

RMA Number:

Refund Check Payable To:

Return Shipping Instructions

You must package the software diskettes with the manual and write the RMA number on the outside of the shipping carton. Shipping charges incurred will not be reimbursed. Ship to:

Lahey Computer Systems, Inc.
865 Tahoe Blvd.
P.O. Box 6091
Incline Village, NV 89450-6091

INDEX

Symbols

.MOD filename extension 13

Numerics

386LINK environment variable 37
386LINK.EXE 12

A

a runtime option 148
AMEDIT 102
-ap switch, arithmetic precision 18
API
 Windows 48
AUTOMAKE 101
 CHECK= 111
 COMPILE= 107
 DEBUG 111
 FILES= 107
 LATESCAN 111
 LF90 107
 LINK= 109
 MAKEMAKE 111
 NOQUITONERROR 111
 OBJDIR= 110
 OBJEXT= 110
 QUITONERROR 111
 SYSINCLUDE= 110
 TARGET= 109
AUTOMAKE configuration file
 editor 102

B

-BANNER, Linker banner
 switch 19
-block, blocksize switch 19
blocks in ED for Windows 58
Borland C++ 39, 42
Borland Delphi 39, 47
breakpoints 63, 69
bugs 154

C

C Compiler User's Guide 9
C runtime option 146

-c, suppress linking switch 19
case conversion 59
CFIG386.EXE 133
-chk, checking switch 19
-chkglobal, global checking switch 21
-co, display compiler options switch 21
code completion 59
coding shortcuts 58
command files 14
 LM 99
compiler 12, 17
 controlling 17
 errors 17
 switches 17
compiling from ED for Windows 60
configuration of ED 66
Conflicts 14
console mode 35
-cover, generate coverage information
 switch 21
Coverage Tool
 -cover switch 21
creating a file 54

D

d runtime option 148
-dal, deallocate allocatables switch 21
-dbl, double switch 21
debugger 11
debugging
 from ED 61
 restrictions 93
 with FDB 67, 115, 125
 with WinFDB 93
DEMO.F90 6
direct file format 139
disassembly 76
distribution 8
divide-by-zero 33
-dll, dynamic link library switch 22
DLL_EXPORT 41
DLL_IMPORT 40
DLLs 12
driver 11
dummy argument 138

dynamic link libraries 12

E

e runtime option 148
ED for Windows 51, 67, 81, 115, 125
 blocks 58
 case conversion 59
 changing compiler options 60
 code completion 59
 coding shortcuts 58
 compiling 60
 configuration 66
 create file 54
 debugging from 61
 editing 57
 exiting 52
 extended characters 57
 find 57
 function/procedure list 56
 help 54
 managing files 54
 matching parentheses and
 statements 57
 menu bar 52, 84
 navigation 56
 opening a file 55
 previous/next procedure 56
 screen 52
 set up 51, 81
 smartype 59
 starting 51, 81
 status bar 53
 templates 58
 toolbar 53
 undo and redo 57
 window bar 54
editing 57
editor 11
 Lahey ED for Windows 51, 67,
 81, 115, 125
efficiency considerations 137
environment variables
 386LINK 37
 FORT90L 144
 FUnn 151

FUnnBF 152
 ERRATA.TXT 9
 errors
 compiler 17
 -Exe, executable file name
 switch 29
 exiting ED for Windows 52
 extended characters 57

F
 f90SQL Lite Help 9
 -f90sql, f90SQL Lite switch 22
 file formats
 direct 138
 formatted sequential 138
 transparent 138
 unformatted sequential 138

-file, filename switch 22
 FILELIST.TXT 9
 filenames 12
 .MOD extension 13
 default linker extensions 37
 object file 13
 output file 13
 source file 13

files
 386LINK.EXE 12
 CFIG386.EXE 133
 ERRATA.TXT 9
 HDRSTRIP.F90 134
 LINKERR.TXT 9
 PENTEST.F90 134
 SEQUNF.F90 134
 TELLME.EXE 134
 TRYBLK.F90 134
 UNFSEQ.EXE 135
 WHICH.EXE 135

find text 57
 -fix, fixed source-form switch 23
 formatted sequential file
 format 138
 FORT90L environment
 variable 144
 -FULLWARN, linker warning
 switch 34
 function/procedure list 56
 FUnn environment variable 151
 FUnnBF environment
 variable 152

G
 g runtime option 148
 -g, debug switch 23

H
 HDRSTRIP.F90 134
 help
 ED for Windows 54
 hints
 determining load image size 139
 efficiency considerations 137
 file formats 138
 performance considerations 139
 side effects 137
 hours 153

I
 i runtime option 149
 -i, include path switch 23
 -implib, DLL library switch 24
 import librarian 12
 -import, import DLL function
 switch 24
 -in, IMPLICIT NONE switch 24
 -info, display informational messages
 switch 25
 installation 3
 invalid operation 33

L
 Lahey ED for Windows 51, 67, 81,
 115, 125
 Lahey Fortran 90 Reference Manual 9
 -li, Lahey intrinsic procedures
 switch 25
 -Lib, library files switch 25
 -LIBPath, library path switch 25, 37
 Librarian
 /EXTRACTALL 97
 /Help 98
 /Pagesize 98
 Syntax 97
 librarian 11, 12, 97
 library manager 97
 library searching rules 37
 limits of operation 141
 linker 12
 default filename extensions 37
 library searching rules 37

 rules 37
 linker search rules 37
 LINKERR.TXT 9
 linking 37
 LM 97
 command files 99
 response files 99
 LM librarian
 command-line syntax 97
 load image size 139
 -long, long integers switch 26
 -lst, listing switch 26

M
 M runtime option 146
 m runtime option 150
 MAKE utility 101
 make utility 11
 -Map, linker map file switch 27
 -MAPNames, linker map symbol
 name length switch 27
 -MAPWidth, linker map file width
 switch 28
 matching parentheses and
 statements 57
 -maxfatals, maximum fatal errors
 switch 28
 menu bar 52, 84
 Microsoft Visual Basic 39, 45
 Microsoft Visual C++ 39, 42
 Mixed 38
 -ml, mixed language switch 28, 40
 ML_EXTERNAL 49
 -mldefault, mixed language default
 switch 28
 -mod, module path switch 29

N
 n runtime option 150
 network installation 3
 notational conventions 2

O
 -o, object file name switch 29
 -o0, optimization level zero switch 29
 -o1, optimization level one switch 29
 object filenames 13
 -OneCase, case insensitive switch 34
 OpenGL graphics 50

- opening a file 55
- Optimization 29
- out, output file switch 29
- output filenames 13
- overflow 33
- P**
- p runtime option 150
- pause, pause after program
 - completion 30
- pca, protect constant arguments
 - switch 30
- PENTEST.F90 134
- preconnected units, standard i/
 - o 145
- previous/next procedure 56
- private, module accessibility
 - switch 30
- program size 141
- programming hints 137
- PUBList, public symbol ordering
 - switch 31
- Q**
- Q runtime option 146
- q runtime option 150
- quad, quadruple precision
 - switch 31
- R**
- r runtime option 151
- Re runtime option 147
- README.TXT 8
- README_API.TXT 8
- README_ASSEMBLY.TXT 8
- README_C.TXT 8
- README_COMPATIBLE.TXT 9
- README_F90GL.TXT 9
- README_F90SQL.TXT 9
- README_SERVICE_ROUTINES.TXT 9
- README_WISK.TXT 8
- registering 2
- registers, displaying in WinFDB 91
- requirements
 - system 1
- Resource Compiler 12
- response files
 - LM 99
- restrictions, debugging 93
- return codes 15
- return values, execution 145
- Ri runtime option 147
- Rm runtime option 147
- runtime options
 - a 148
 - C 146
 - d 148
 - e 148
 - g 148
 - i 149
 - M 146
 - m 150
 - n 150
 - p 150
 - Q 146
 - q 150
 - r 151
 - Re 147
 - Ri 147
 - Rm 147
 - Ry 147
 - T 147
 - u 151
 - x 151
- runtime options, syntax 146
- Ry runtime option 147
- S**
- Sampler Tool
 - trace switch 33
- sav, SAVE local variables switch 31
- scrollable window, -vsw switch 34
- searching rules
 - library 37
- SEQUNF.F90 134
- setting up ED for Windows 51, 81
- side effects 137
- smartype 59
- source filenames 13
- SSL2 Extended Capabilities User's Guide 9
- SSL2 Extended Capabilities User's Guide II 9
- SSL2 User's Guide 9
- Stack, stack size switch 32
- standard input/output units 145
- starting ED for Windows 51, 81
- static linking 49
- staticlink, static linking switch 32
- status bar 53
- stchk, stack overflow check
 - switch 32
- step 63, 71
- step into 63
- step over 63
- support services 143, 153
- switches
 - ap, arithmetic precision 18
 - block, blocksize 19
 - c, suppress linking 19
 - changing in ED 60
 - chk, checking 19
 - chkglobal, global checking 21
 - co, display compiler options 21
 - compiler 17
 - cover, generate coverage information 21
 - cross-reference listing 36
 - dal, deallocate allocatables 21
 - dbl, double switch 21
 - description 13
 - dll, dynamic link library 22
 - f90sql, f90SQL Lite 22
 - file, filename 22
 - fix, fixed source-form 23
 - g, debug 23
 - i, include path 23
 - implib, DLL library 24
 - import, import DLL function 24
 - in, IMPLICIT NONE 24
 - info, display informational messages 25
 - li, Lahey intrinsic procedures 25
- linker
 - BANNER, Linker banner 19
 - Exe, executable file name 29
 - FULLWARN, linker warning 34
 - Lib, library files 25
 - LIBPath, library path 25, 37
 - Map, map file 27
 - MAPNames, map symbol

- name length 27
 - MAPWidth, map file width 28
 - OneCase, case insensitive 34
 - PUBList, public symbol ordering 31
 - Stack, stack size 32
 - TwoCase, case sensitive 34
 - long, long integers 26
 - lst, listing 26
 - maxfatals 28
 - ml, mixed language 28
 - mldefault, mixed language default 28
 - mod, module path 29
 - o, object file name 29
 - o0, optimization level zero 29
 - o1, optimization level one 29
 - out, output file 29
 - pause, pause after program completion 30
 - pca, protect constant arguments 30
 - private, module accessibility 30
 - quad, quadruple precision 31
 - sav, SAVE local variables 31
 - staticlink, static linking 32
 - stchk, stack overflow check 32
 - swm, suppress warning messages 32
 - t4, target 486 33
 - tp, target Pentium 33
 - tpp, target Pentium Pro 33
 - trace, location and call traceback for runtime errors 33
 - trap, trap NDP exceptions 33
 - vsw, very simple windows 34
 - w, warn 34
 - WARN, linker warning 34
 - win, Windows 35
 - wisk, Winteracter Starter Kit 35
 - wo, warn obsolescent zero 36
 - swm, suppress warning message(s) switch 32
 - syntax
 - LM command-line 97
 - syntax highlighting 56
 - system requirements 1
- T**
- T runtime option 147
 - t4, target 486 switch 33
 - technical support 154
 - Technical Support Questionnaire 155
 - TELLME.EXE 134
 - templates 58
 - toolbar 53
 - tp, target Pentium switch 33
 - tpp, target Pentium Pro switch 33
 - trace, location and call traceback for runtime errors switch 33
 - transparent file format 139
 - trap, trap NDP exceptions switch 33
 - TRYBLK.F90 134
 - TwoCase, case sensitive switch 34
- U**
- u runtime option 151
 - underflow 33
 - undo and redo 57
 - unformatted sequential file format 138
 - UNFSEQ.EXE 135
- V**
- Visual Analyzer User's Guide 9
 - vsw, very simple windows switch 34
- W**
- w, warn switch 34
 - WARN, linker warning switch 34
 - Warranties 155
 - warranties 155
 - watch dialog 65
 - WHICH.EXE 135
 - win, Windows switch 35
 - winconsole, Windows console-mode switch 35
 - window bar 54
 - window, scrolling, -vsw switch 34
 - Windows 7, 35
 - Windows API 48
 - Windows console-mode 35
 - WinFDB 81
 - command line entry 92
 - load map display 92
 - registers display 91
 - restrictions 93
 - traceback display 91
 - watch window 90
 - Winteracter Starter Kit 7, 9
 - WISK 7
 - WiSK Help 9
 - wisk, Winteracter Starter Kit switch 35
 - wo, warn obsolescent switch 36
 - World Wide Web 155
- X**
- x runtime option 151
 - xref, cross-reference listing switch 36
- Y**
- Y2K compliance, Ry runtime option 147
- Z**
- zero, initialize variables to zero switch 36