LF Fortran Express User’s Guide

Revision D

Copyright

Trademarks

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)
support2@lahey.com (all versions)

Table of Contents

Command-Line Debugging with FDB. 95

1 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, linker, debugger and librarian.

LF95 includes two manuals: the *Express User's Guide* (this book), which describes how to use the tools; and the *Language Reference,* which describes the Fortran 95 language.

Manual Organization

This book is organized into six chapters and three 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, *Mixed Language Programming*, describes building statically linked and dynamically linked mixed language applications, and discusses interfacing Fortran procedures with procedures written with other languages.
- Chapter 4, *Command-Line Debugging with FDB,* describes the command-line debugger.
- Chapter 5, *Library Manager,* describes command-line operation of the librarian.
- Chapter 6, *Utility Programs*, describes how to use the additional utility programs.
- Appendix A, *Programming Hints*, offers suggestions about programming in Fortran on the PC with LF95.
- Appendix B, *Runtime Options*, describes options that can be added to your executable's command line to change program behavior.

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.

A vertical bar separating non italic characters enclosed in curly braces '{ opt1 | opt2 | opt3 }' indicates a set of possible options, from which one is to be selected.

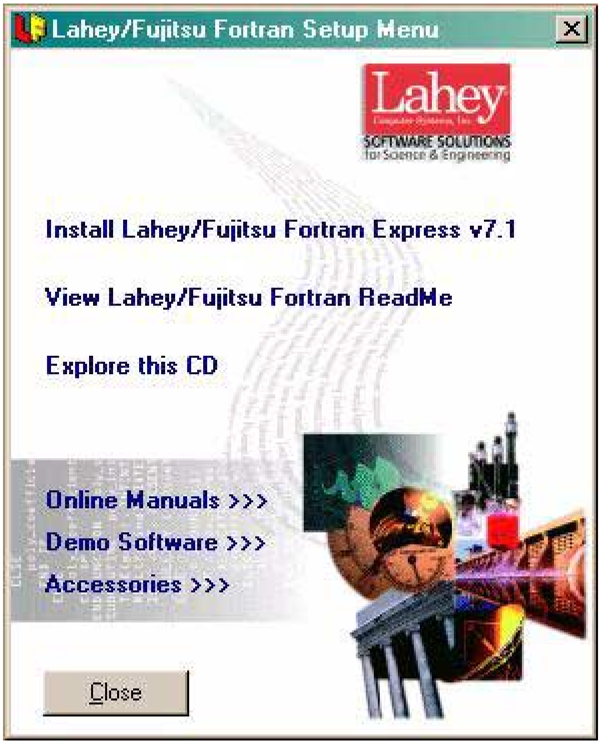
System Requirements

- A Pentium series or compatible processor
- 32 MB of RAM
- 50 MB of available hard disk space for typical installation
- Windows 98, Windows ME, Windows NT 4.0, Windows 2000, or Windows XP.

Installing Lahey/Fujitsu Fortran 95

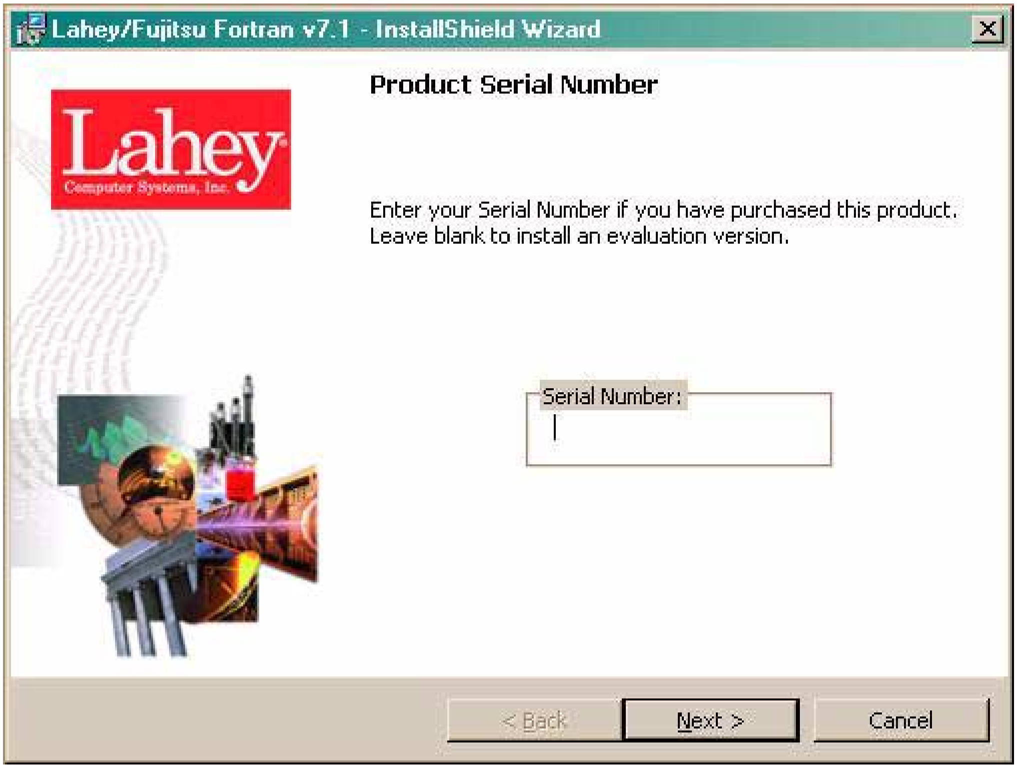
Before starting, review the System Requirements. Administrator rights are required for installation.

1. **Insert disk – Lahey/Fujitsu Fortran Express v7.1**
   The following Lahey/Fujitsu Fortran Setup Menu will automatically display when the CD is inserted in the drive. If the Setup Menu does not display, run d:\LFSetup.exe, where d: is your CD drive..


Lahey/Fujitsu Fortran Setup Menu
Lahey
Computer Systems, Inc.
SOFTWARE SOLUTIONS
for Science & Engineering
Install Lahey/Fujitsu Fortran Express v7.1
View Lahey/Fujitsu Fortran ReadMe
Explore this CD
Online Manuals >>>
Demo Software >>>
Accessories >>>
Close

2. **Choose Install Lahey/Fujitsu Fortran Express v7.1 from the Setup Menu.**

a. The following dialog will appear.



Lahey/Fujitsu Fortran v7.1 - InstallShield Wizard

Product Serial Number

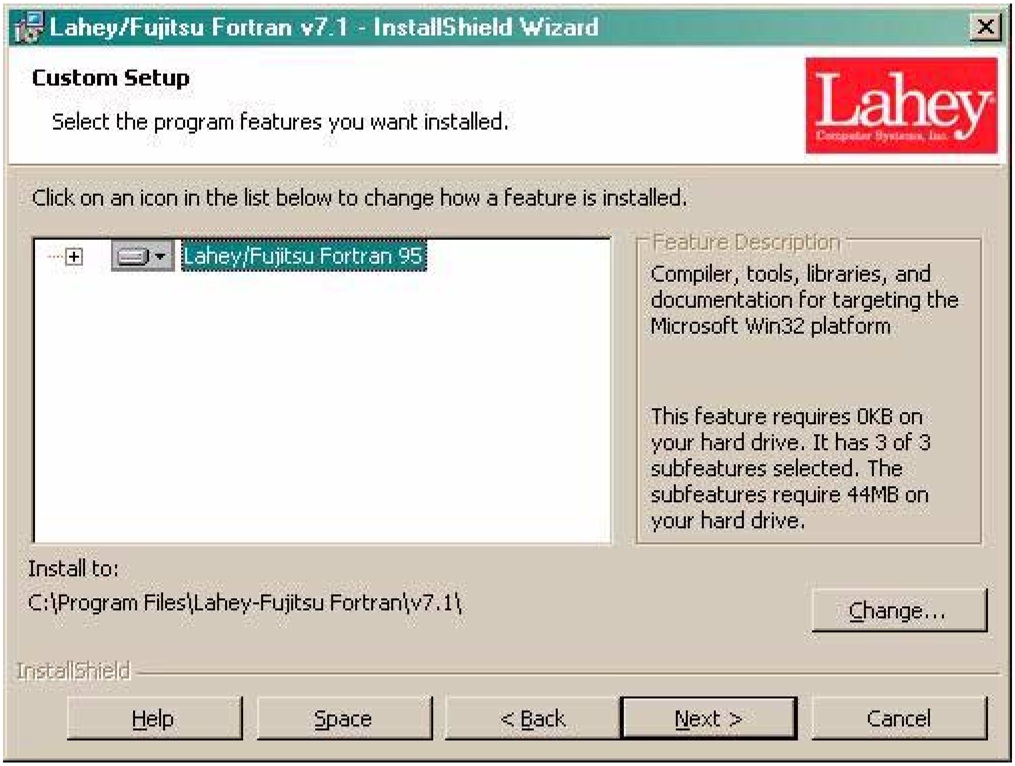
Enter your Serial Number if you have purchased this product.
Leave blank to install an evaluation version.

Serial Number:

Enter your serial number if you purchased this product, or leave blank to install the evaluation version. If you install the evaluation version at this time, you can convert it into a licensed version any time after purchase (see Product License Activation). A serial number is required to receive technical support.

b. Follow the prompts to install this product.

c. If desired, choose Custom on the Setup Type dialog to change the installation folder or to add or remove specific features. The following picture shows the features available for the Enterprise Edition:



Lahey/Fujitsu Fortran v7.1 - InstallShield Wizard

Custom Setup

Select the program features you want installed.

Click on an icon in the list below to change how a feature is installed.

Lahey/Fujitsu Fortran 95

Feature Description

Compiler, tools, libraries, and documentation for targeting the Microsoft Win32 platform

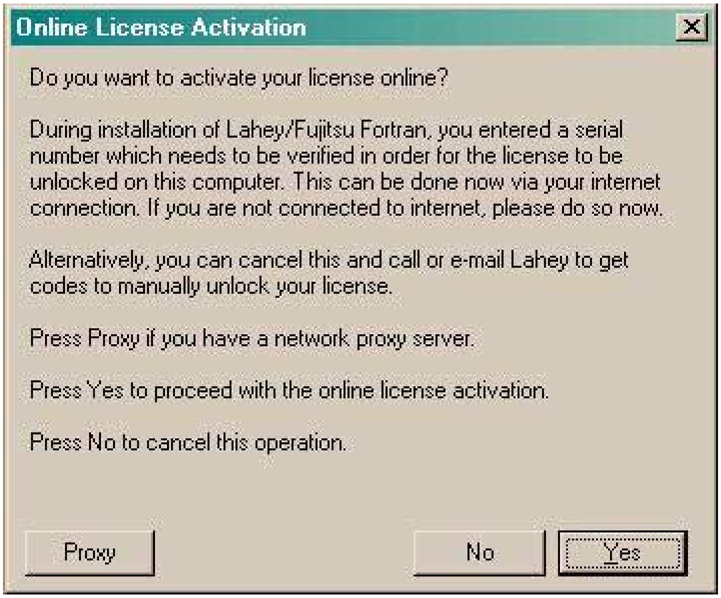
This feature requires 0KB on your hard drive. It has 3 of 3 subfeatures selected. The subfeatures require 44MB on your hard drive.

Install to:

C:\Program Files\Lahey-Fujitsu Fortran\v7.1\

d. Select Finish when setup is complete.

e. If you entered a serial number at the beginning of the setup, you will be given the following choice to activate your license online at this time:



Online License Activation

Do you want to activate your license online?

During installation of Lahey/Fujitsu Fortran, you entered a serial number which needs to be verified in order for the license to be unlocked on this computer. This can be done now via your internet connection. If you are not connected to internet, please do so now.

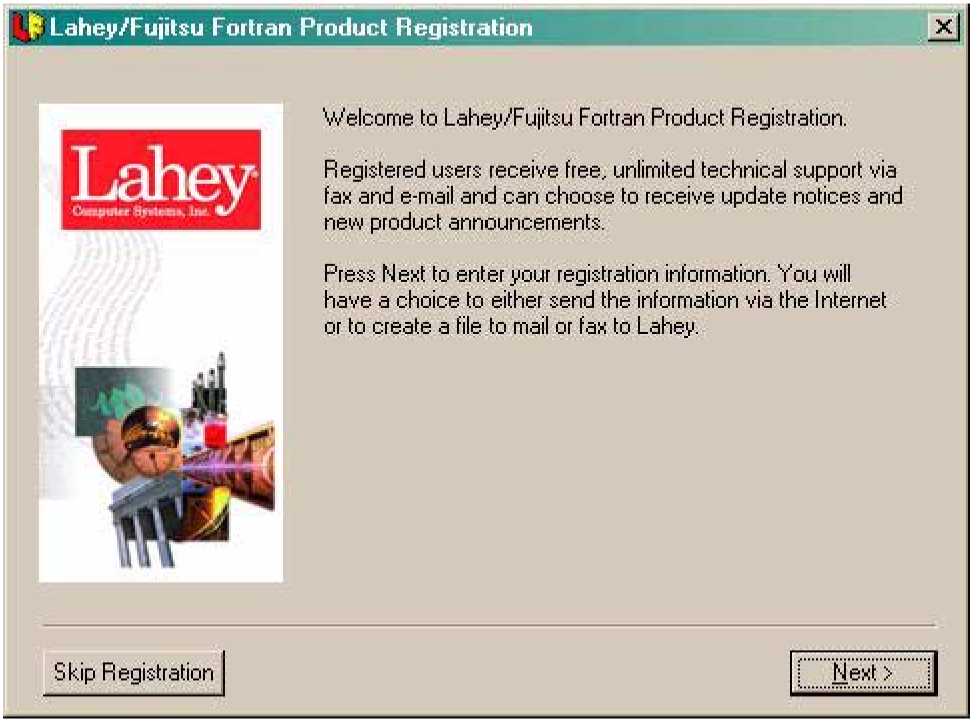
Alternatively, you can cancel this and call or e-mail Lahey to get codes to manually unlock your license.

Press Proxy if you have a network proxy server.

Press Yes to proceed with the online license activation.

Press No to cancel this operation.

If online license activation is successful, you will be given the choice to register your product with Lahey online. You can choose to send the registration online, or create a file with registration information to send to Lahey.



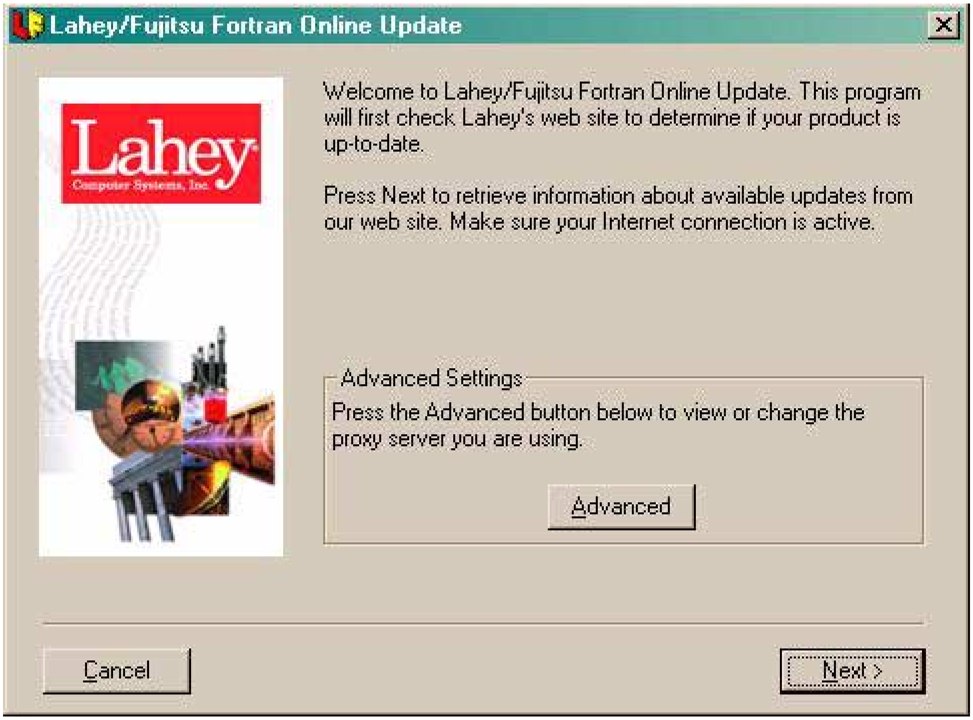
Lahey/Fujitsu Fortran Product Registration

Welcome to Lahey/Fujitsu Fortran Product Registration.

Registered users receive free, unlimited technical support via fax and e-mail and can choose to receive update notices and new product announcements.

Press Next to enter your registration information. You will have a choice to either send the information via the Internet or to create a file to mail or fax to Lahey.

g. Finally, if online activation was successful, you will be given the choice to check for product updates that may be available on Lahey's website.



Lahey/Fujitsu Fortran Online Update

Welcome to Lahey/Fujitsu Fortran Online Update. This program will first check Lahey's web site to determine if your product is up-to-date.

Press Next to retrieve information about available updates from our web site. Make sure your Internet connection is active.

Advanced Settings

Press the Advanced button below to view or change the proxy server you are using.

The appropriate LF95 directory names are appended to the PATH, LIB, and INCLUDE environment variables. The variables are appended, rather than prepended, in order to be less obtrusive to your current configuration. For the compiler, tools and utilities that are used as command-line programs, the "LF95 Console Prompt" is available on the Programs menu to start a console command-line with the environment variables optimally set. To to ensure correct operation of compilers, tools, and utilities in this product, we recommend either using the LF95 console prompt, or editing the aforementioned environment variables to put the LF95 directories ahead any others.

If you are using Windows 2000 or XP, your installation is complete. Otherwise, reboot your system (or log out and log in if using Windows NT) to insure that your system environment are properly set. You are now ready to build your first program.

License Activation

By default, the Lahey/Fujitsu Fortran v7.1 product is installed with licenses set up to expire after an evaluation period. To continue using a purchased product, the licenses must be activated. Activation is done after a product is purchased or upgraded. The License Activation program cannot be used to purchase a product.

When license activation takes place, it will only be valid on the PC on which it was activated. There are several scenarios that will cause the license to revert to a trial version:

- This product is copied, moved, or installed on another PC.
- The hardware configuration of the PC is significantly changed.
- The Windows operating system is reinstalled.
- The Windows registry is reverted to a version older than the time of activation.

If you uninstall and reinstall this product on the same PC configuration, it will still be activated. Contact sales@lahey.com when you need to reactivate your purchased license on a different PC.

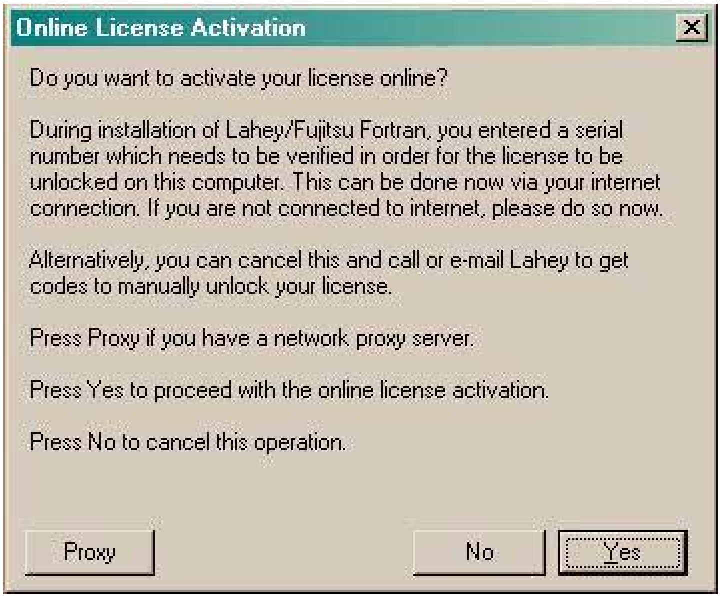
License Activation

Product activation can be accomplished in several ways:

1. The serial number for a purchased product is entered at the beginning of the Lahey/ Fujitsu Fortran installation, and online activation takes place after the installation is complete.
2. The License Activation program is run and a purchased product's serial number is entered for online activation.
3. The License Activation program is run and manual codes are entered after receiving them from Lahey.

Activation During Installation

If you entered a serial number at the beginning of the setup, at the end you will be given the following choice to activate your license online:



Online License Activation

Do you want to activate your license online?

During installation of Lahey/Fujitsu Fortran, you entered a serial number which needs to be verified in order for the license to be unlocked on this computer. This can be done now via your internet connection. If you are not connected to internet, please do so now.

Alternatively, you can cancel this and call or e-mail Lahey to get codes to manually unlock your license.

Press Proxy if you have a network proxy server.

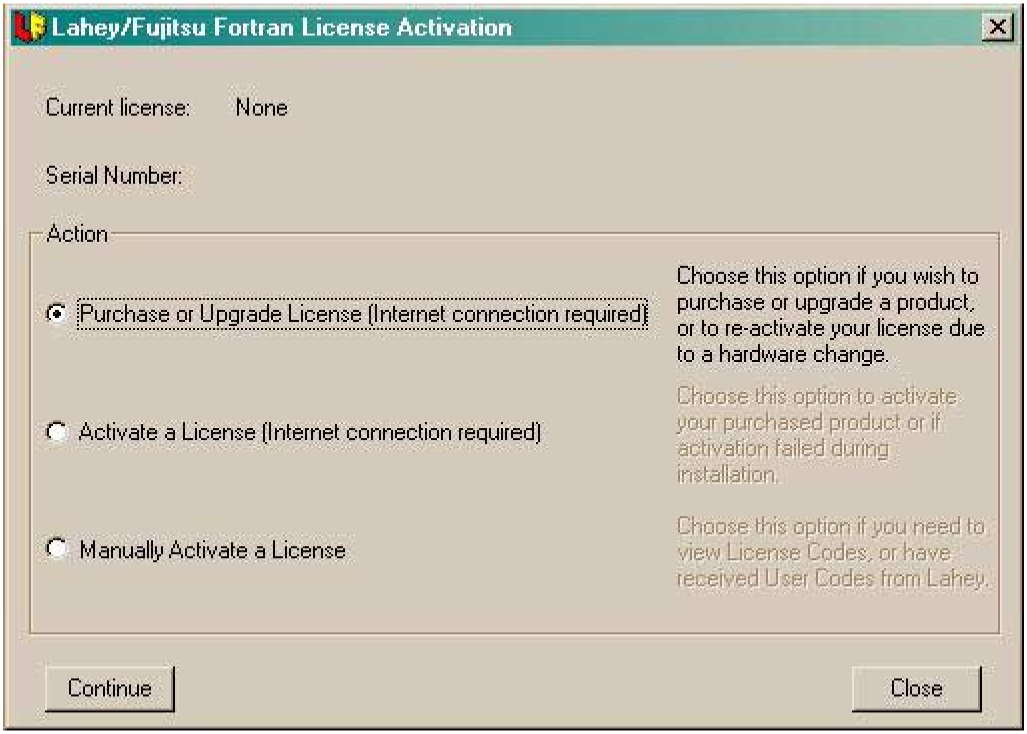
Press Yes to proceed with the online license activation.

Press No to cancel this operation.

Before pressing Yes, be sure to press Proxy and enter your proxy address if you have a network proxy server. When Yes is pressed, the online activation will be initiated, and you will be notified whether activation was successful. If successful, your product is ready to use. If No is pressed, or the online activation fails, you will have to activate the product by using the License Activation utility program, as described below.

The License Activation Utility Program

Select the License Activation shortcut from the Programs menu, under Lahey/Fujitsu Fortran v7.1, Product Maintenance. The following dialog will appear:



Lahey/Fujitsu Fortran License Activation

Current license: None

Serial Number:

Action

● Purchase or Upgrade License (Internet connection required)

Choose this option if you wish to purchase or upgrade a product, or to re-activate your license due to a hardware change.

○ Activate a License (Internet connection required)

Choose this option to activate your purchased product or if activation failed during installation.

○ Manually Activate a License

Choose this option if you need to view License Codes, or have received User Codes from Lahey.

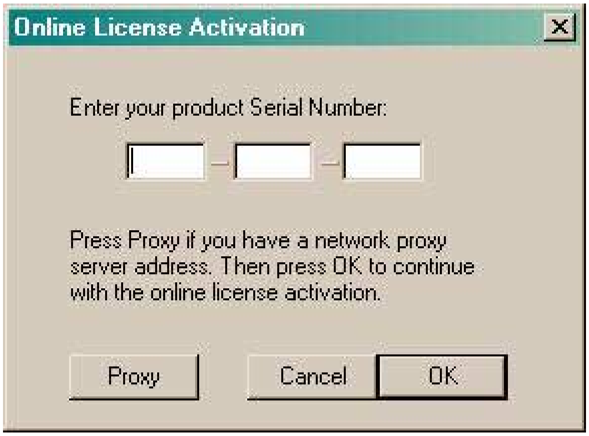
Purchase or Upgrade License

If you have not purchased the product or wish to purchase an upgrade to a new edition, choose this option and press Continue. This will display a web page at www.lahey.com with further instructions for purchasing.

Also, choose this option if you purchased the product and already have a serial number, but wish to manually activate a license. The web page contains instructions.

Activate or Reactivate a License

If you have a purchased product serial number, choose this option to activate your license online. Press Continue and this dialog will appear:



Online License Activation

Enter your product Serial Number:

Press Proxy if you have a network proxy server address. Then press OK to continue with the online license activation.

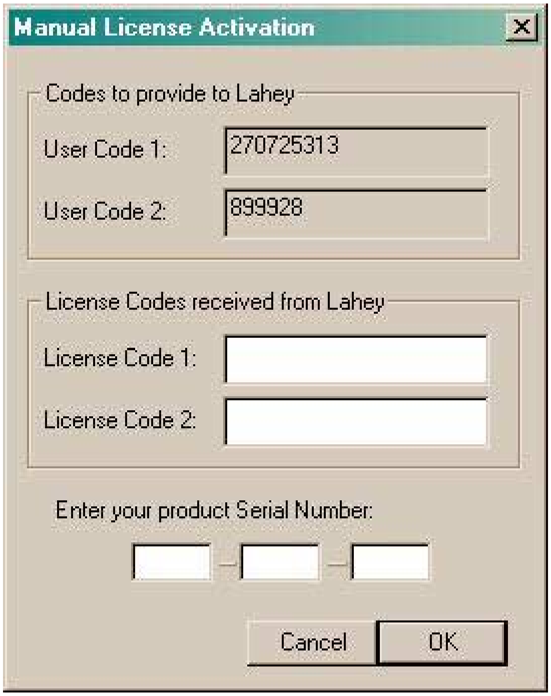
Before pressing Yes, be sure to press Proxy and enter your proxy address if you have a network proxy server. When you press OK, activation will be attempted over the internet.

Manually Activate a License

This option is used for several purposes:

- Online activation is not desired, or not possible.
- Your license needs reactivation, perhaps due to a PC change.
- A special function needs to be performed on your license.

After pressing Continue, the following dialog will appear:


Manual License Activation
Codes to provide to Lahey
License Codes received from Lahey
License Code 1:
License Code 2:
Enter your product Serial Number:
Cancel
OK

User Code 1: 270725313

User Code 2: 899928

(The User Code numbers above are examples only.)

User Code 1 and User Code 2 are numbers that need to be provided to Lahey Sales or Technical Support personnel before the License Code(s) can be given to you. Once you receive the License Code(s), enter them and your serial number in the dialog fields and press OK.

Note that you can press cancel after copying down the User Codes to provide to Lahey, and rerun the License Activation utility program at a later time to enter the License Codes.

After Activation

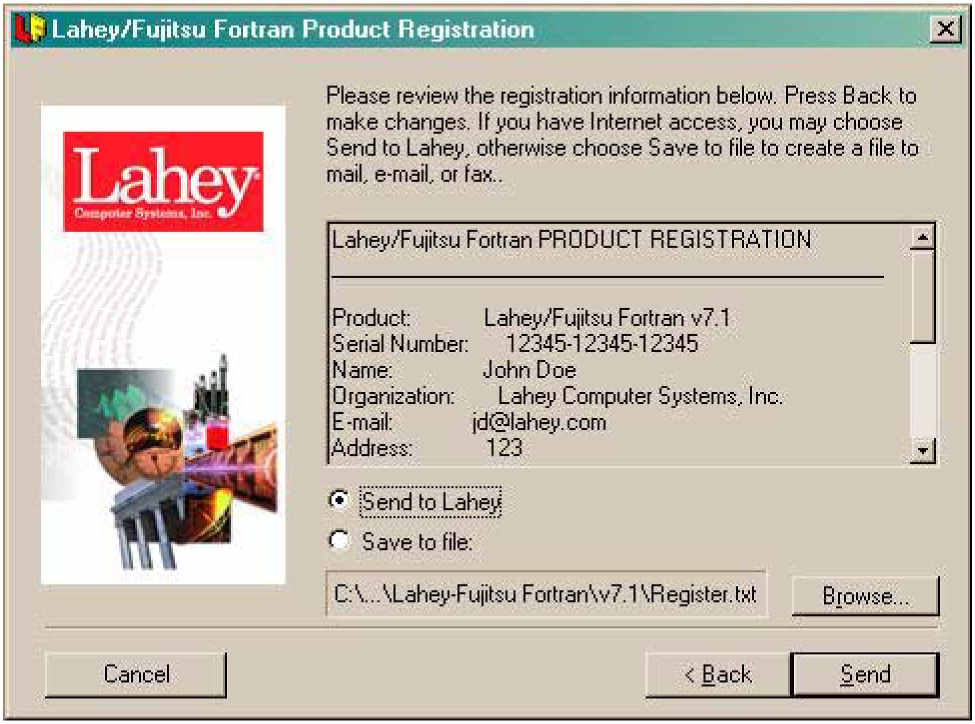
When a new activation has taken place, you will be presented with the choices to register your product with Lahey and to check for available product updates.

An upgrade to a new edition can be accomplished using the License Activation utility program – the Enterprise edition will be issued a new serial number at time of purchase.

Product Registration

Please register your Lahey product. When you activate your product license, you will be prompted to register. Also, you can initiate the registration program by selecting the Product Registration shortcut from the Program menu, or register at our website, www.lahey.com.

If you move or transfer a Lahey product's ownership, please let us know.



Lahey/Fujitsu Fortran Product Registration

Please review the registration information below. Press Back to make changes. If you have Internet access, you may choose Send to Lahey, otherwise choose Save to file to create a file to mail, e-mail, or fax..

```
Lahey/Fujitsu Fortran PRODUCT REGISTRATION

Product:        Lahey/Fujitsu Fortran v7.1
Serial Number:    12345-12345-12345
Name:           John Doe
Organization:     Lahey Computer Systems, Inc.
E-mail:         jd@lahey.com
Address:         123
```

● Send to Lahey

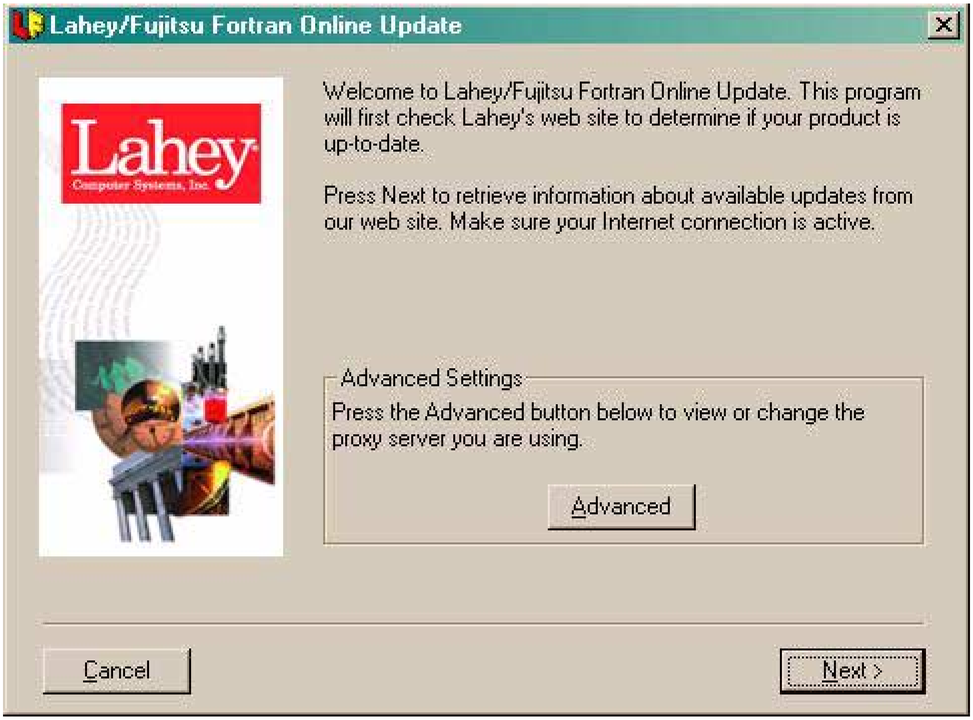
○ Save to file:

C:\...\Lahey-Fujitsu Fortran\v7.1\Register.txt

Maintenance Updates

Maintenance updates for purchased products are made available from Lahey's website. They comprise bug fixes or enhancements or both for this version of this product. The update program applies "patches" to your files to bring them up-to-date. The maintenance update version shows as the last two-digits of the version of your compiler. This is displayed in the first line of output when you run the compiler

Any time you want to check the availability of a maintenance update for this version, select Online Update from the Programs menu, and a program will step you through the updating process:



Welcome to Lahey/Fujitsu Fortran Online Update. This program will first check Lahey's web site to determine if your product is up-to-date.

Press Next to retrieve information about available updates from our web site. Make sure your Internet connection is active.

Advanced Settings

Press the Advanced button below to view or change the proxy server you are using.

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 patch files will be downloaded and applied. You need to be connected to the Internet to perform the check and download the files.

To automatically check for updates at regular intervals at startup, press the Advanced button after starting Online Update and enter the interval (in days) in the Check Interval field. An LF Online Update icon will be added to your Windows Startup folder. At startup, Online

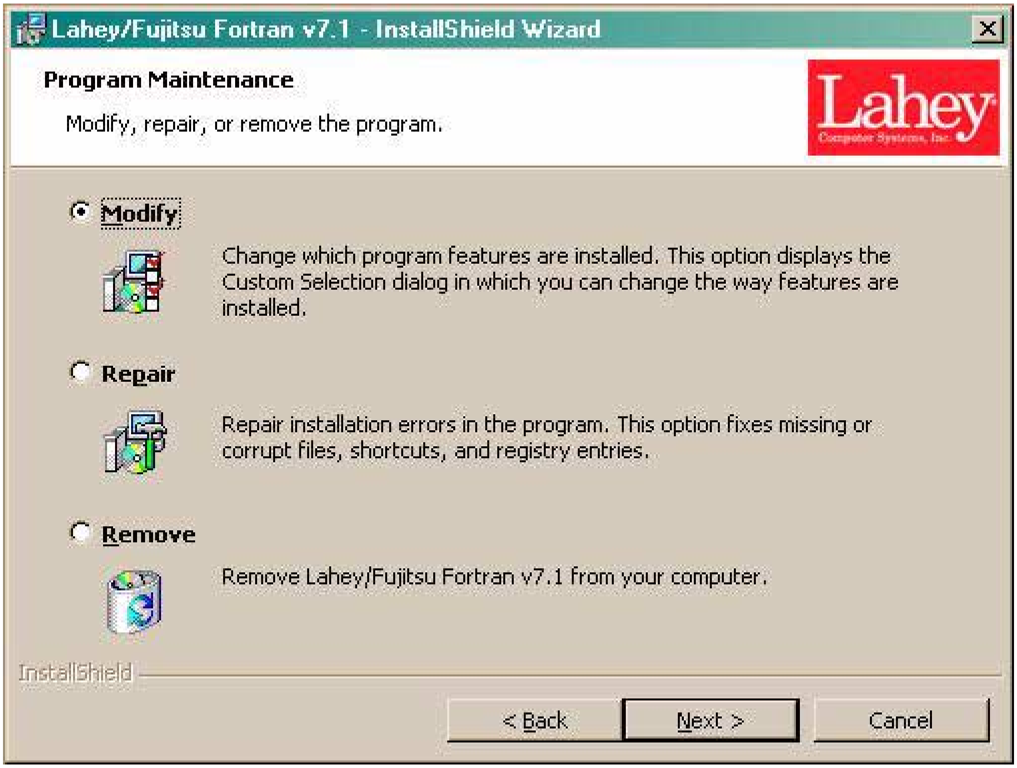
Update will start only if the specified check interval days have passed since the last time Online Update was run. Thereafter, to disable automatic checking, set the check interval to 0 (zero) days.

Another way to get the latest maintenance update for this version is by going to Lahey's web site at www.lahey.com and navigate to Downloads. 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 on systems that are not connected to the Internet, or who want the ability to revert to a previous maintenance version by saving the update programs.

In general, if you modify the contents of any of the files installed by this product (except within the Examples directory), that particular file will no longer be valid for updating, and the update installation program may abort with an error message.

Repairing LF95

The repair program can be found in the Add/Remove Programs applet in the system Control Panel. Select Lahey/Fujitsu Fortran v7.1 and press the Change button. The Windows installer will launch a program and you will see this dialog:



Lahey/Fujitsu Fortran v7.1 - InstallShield Wizard

Program Maintenance

Modify, repair, or remove the program.

● **Modify**

Change which program features are installed. This option displays the Custom Selection dialog in which you can change the way features are installed.

○ **Repair**

Repair installation errors in the program. This option fixes missing or corrupt files, shortcuts, and registry entries.

○ **Remove**

Remove Lahey/Fujitsu Fortran v7.1 from your computer.

Choosing Modify allows you to change which program features are installed.

Choosing Repair will run through the original installation and fix missing or corrupt files, shortcuts, and registry entries.

You can also uninstall the product by choosing Remove.

Uninstalling LF95

To completely remove the Lahey/Fujitsu Fortran product installation, open the Add/Remove Programs applet in the system Control Panel. Select Lahey/Fujitsu Fortran v7.1 and press the Remove button. You will be prompted to confirm the removal, then the uninstall program will continue.

Building Your First LF95 Program

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 v7.1|LF Console Prompt. From the command prompt, build the demo program by changing to LF95's `EXAMPLES` directory (where `DEMO.F90` is installed), and entering

```
LF95 demo
```

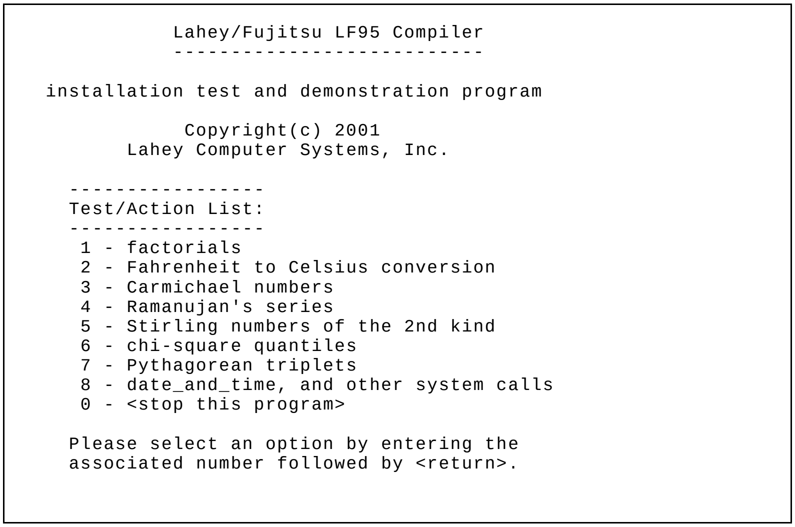
This causes the compiler to read the source file `DEMO.F90` (the extension `.F90` is assumed by default) and compile it into the object file `DEMO.OBJ`. Once `DEMO.OBJ` is created, `LF95` invokes the linker to combine necessary routines from the runtime library and produce the executable program, `DEMO.EXE`.

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/Fujitsu LF95 Compiler
              ---------------------------

   installation test and demonstration program

               Copyright(c) 2001
          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.

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

`README.HTM`	last-minute information
`README_PORT_LF90.TXT`	information on porting your code from LF90
`README_PORT_56.TXT`	information on porting your code from LF95 v5.x
`README_SERVICE_ROUTINES.TXT`	POSIX and other service routines

`IO_ERROR.TXT`	runtime I/O error messages
`RTERRMSG.TXT`	other runtime error messages

Manuals

Lahey/Fujitsu Fortran 95 Language Reference
Lahey/Fujitsu Fortran 95 Express User’s Guide (this document)

Newsletters

The Lahey Fortran *Source* newsletter

Lahey Web Page

`http://www.lahey.com`

Technical Support

For the most up to date support information, please visit the support page at Lahey's website: 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. LF95 controls compilation, the production of executable programs, static link libraries, and dynamic link libraries (DLLs).

The Development Process

Developing applications with LF95 involves the following tools:

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 a compiler, but it is actually a driver that invokes the appropriate compiler, linker, and other components used to create executables, libraries, and other products.

Library Manager. Use the library manager to create, change, and list the contents of object libraries. See Chapter 5, *Library Manager*, for instructions on how to use the library manager.

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

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. Supplemental processes, like creating static libraries, DLL's, 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. `LIB.EXE` is the library manager. It can be invoked from the driver or from the command prompt to create or change static libraries.

Linker. `LINK.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, and 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 can be sent to the linker, or can be converted by `CVTRES.EXE` into object files.

Running LF95

By default, the LF95 driver program will compile any specified source files and link them along with any specified object files and libraries into an executable program.

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

LF95 *filenames [options]*

The driver searches for the various tools (the compiler, library manager, linker, and resource compiler) first in the directory the driver is located and then, if not found, on the DOS path.

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

The command line options 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 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 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 an option specifying otherwise:

`.F90` and `.F95` specifies interpretation as Fortran 95 free source form.

`.FOR` and `.F` specify interpretation as Fortran 95 fixed source form.

Source files for a given invocation of the driver should not mix free form and fixed form. If files with both the `.FOR` or `.F` and `.F90` or `.F95` 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 options can be used to control the assumed extension and override the interpretation specified by the extension. see *“-[N]FIX”* on page 35

Object Filenames

The default name for an object file is the same as the source file name. By default, the object file is placed in the current directory.

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. By default, output files are placed in the same directory as the first file encountered. This may be overridden by specifying the `-OUT` option with a new path and name (see *“-OUT filename”* on page 40). The default extension for executable files is `.EXE`. The default extension for static link libraries is `.LIB`. The default extension for dynamic link libraries is `.dll`.

Options

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

Some options 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 option is detected, the entire text from the beginning of the unknown option to the beginning of the next option or end of the command line is passed to the linker.

Conflicts Between Options

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

Driver Configuration File (LF95.FIG)

In addition to specifying options on the command line, you may specify a default set of options in the file `LF95.FIG`. When the driver is invoked, the options in `LF95.FIG` are processed before those on the command line. Command-line options override those in `LF95.FIG`. 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 options 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 the `@` 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 options 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 when the process is complete.

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. If a linker or resource compiler error occurs, LF95 exits with the exit code from the failing tool. Other return codes are listed below:

Table 1: Driver Return Codes

Code	Condition
`0`	Successful compilation and link
`1`	Compiler or tool failed to run or fatal compilation error occurred
`2`	Library Manager error
`4`	Driver error
`5`	Help requested

Note that there may be overlap between exit codes presented in Table 1 and exit codes passed through from a tool.

Creating a Console-Mode Application

LF95 creates Windows console-mode executables by default, so no options 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 W*inter-acter*, GINO, or RealWin) or by calling the Windows API's directly, specify the `-win` option. To call the Windows API's directly, you must also specify the `-ml winapi` option (see *"-ML { bc | bd | fc | lf90 | lf95 | msvb | msvc | winapi }"* on page 39 and *"Calling the Windows API"* on page 78 for more information). Note that console I/O is not permitted when using the `-win` option.

Example

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

Creating a 32-bit Windows DLL

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

Example

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

In this example, the source file `MYPROG.F90` contains procedures 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 *”Dynamically linked applications”* on page 54.

Creating a static library

To create a static library, specify the library name using the `-out` option.

Example

```
LF95 mysub.f90 -out mylib.lib
```

LF95 recognizes that a library is requested because of the .lib extension for the output file. This causes LF95 to invoke the library manager rather than the linker. If the library specified with `-out` does not exist, it is created; if it already exists, it is updated.

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 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

The OpenGL Libraries

To use f90gl/OpenGL you will need three OpenGL DLL's installed in your Windows SYSTEM or SYSTEM32 directory:

OPENGL32.DLL
GLU32.DLL
GLUT32.DLL

The first two of these libraries are a standard part of Windows NT4, 2000, XP, 95(OSR2), 98 and Me. Many video card manufacturers now also provide accelerated OpenGL support as part of their video drivers. These drivers may replace the functionality of these two DLL's.

GLUT32.DLL is not part of the standard Windows distribution. GLUT32.DLL will be installed in the System or System32 (NT) directory by the installation program.

The f90gl Libraries & Modules

The f90gl interface on the f90gl website is posted in source form only. For many users this is unsuitable since it requires a C compiler and a certain level of technical expertise in building the interface. In the case of Lahey LF95, which uses the Microsoft linker, f90gl is best built using Microsoft Visual C.

This product eliminates the need for C compilers by providing pre-built f90gl modules and libraries suitable for use with Lahey LF95 5.7 and newer. The sources for f90gl are not included here since they are not required (as noted, they are available from the f90gl website).

Example Programs

A subset of the f90gl examples are supplied in the LF95 EXAMPLES directory. A RunDemos.bat file is included to build and run all of the examples.

Compilation and linking of f90gl programs simply requires that the LF95 LIB directory be specified in the compiler module path and that the names of the f90gl libraries are specified for linking. Specify -win to create a Windows program. See the RUNDEMOS.BAT file for command line examples. These are substantially simplified from the somewhat complex MF8N?O.BAT equivalents supplied with the f90gl distribution.

Example programs:

- Blender - two rotating objects, one red, one green, which fade in and out, plus some text.
- Fbitfont - some text
- Fscene - three 3D objects in red. The right mouse button brings up a menu. Redraw is really slow in outline mode on some machines.
- Logo - the f90gl logo. Rotate with the mouse while holding the left mouse button. Right mouse button brings up a menu. Middle mouse button selects a new value on the color bars (rgb sliders).
- Modview - contains a module for using the mouse and arrow keys to rotate, zoom, pan and scale. Initially the left button rotates (hold button down while moving mouse), middle button zoom, arrow keys pan, and right button brings up a menu.
- Olympic - the olympic rings come flying into position. Restart the animation with the space bar; terminate with escape.

- Plotfunc - plots a function of two variables as contours, a surface mesh, or a solid surface. Uses the modview module. Right button brings up a menu.
- Scube - a rotating cube in front of a background. Right mouse button brings up a menu. There are also keyboard keys for the same functions as on the menu (look for keyboard in the source code).
- Sphere - a red sphere.

Sources of Information

General inquiries and bug reports regarding f90gl should be sent to:

william.mitchell@nist.gov.

Lahey specific issues should be directed to support2@lahey.com.

OpenGL information can be found at http://www.opengl.org.

Controlling Compilation

During the compilation phase, the driver submits specified source files to the compiler for compilation and optimization. If the `-c` (compile only) option is specified, processing will stop after the compiler runs and modules are created (if necessary). See *”-[N]C”* on page 30. Otherwise, processing continues with the appropriate action depending on what sort of output file is requested.

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 option.

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 option (see *”-[N]INFO”* on page 36).

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 25). Unrecoverable or serious errors detected by the compiler (improper syntax, for example) terminate the build process, and the DOS `ERRORLEVEL` is set to one. An object file is not created.

Compiler and Linker Options

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

Compiling and linking can be broken into separate steps using the -c option. Unless the -c option 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 options to be ignored.

Note also that linker options may be abbreviated as indicated by the uppercase characters in the option name. For example, the `-LIBPath` option can be specified as either `-libpath` or `-libp.` Some linker options 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.

-BLOCK *blocksize*

Default blocksize

Compile only. Default: 8192 bytes

Default to a specific blocksize for file I/O (See the OPEN Statement 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 make files, where it is not always desirable to perform the entire build process with one invocation of the driver.

-*[*N*]*CHK *[***(***[***a***][***,e***][***,s***][***,u***][***,x***]***)***]*

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.

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

Table 3: -chk Arguments

Diagnostic Checking Class	Option 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.

If -chk (a) is specified in conjunction with -pca, the action of -chk (a) is overridden by the action of -pca. In this case, no error is generated when a dummy argument that is associated with a constant actual argument is assigned a new value in the subprogram.

Specifying `-chk (u)` checks for undefined variables by initializing them with a bit pattern. If that bit pattern is detected in a variable on the right side of an assignment then chances are that the variable was uninitialized. Unfortunately, you can get a false diagnostic if the variable holds a value that is the same as this bit pattern. This behavior can be turned off by not using the u argument to the `-chk` option. The values used with `-chk (u)` are:

One-byte integer: -117

Two-byte integer: -29813

Four-byte integer: -1953789045

Eight-byte integer: -8391460049216894069

Default real: -5.37508134e-32

Double precision real: -4.696323204354320d-253

Quadruple precision real: -9.08184876275322841540728989642137 42q-4043

Default complex: (-5.37508134e-32,-5.37508134e-32)

Double precision complex: (-4.696323204354320d-253,-4.696323204354320d-253)

Quadruple precision complex: (-9.08184876275322841540728989642137 42q-4043, -908184876275322841540728989642137 42q-4043)

Character : Z’8B’

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`. Some of the arguments to the `-chk` option may severely impact program execution speed, depending on the source code. Eliminating unneeded options will improve speed.

Example

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

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

The `-chk` option will not check bounds (s) 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 statement or construct.
- The referenced expression has the PARAMETER attribute.
- The parent string is a scalar constant.

Undefined variables (u) are not checked if:

- Subscript checking (s) is also specified, and diagnostic message 0320-w, 0322-w, or 1562-w is issued.
- The referenced expression has the POINTER attribute or is a structure variable one of whose structure components has the POINTER attribute.
- The referenced expression has the SAVE attribute.
- The referenced expression is an assumed-shape array.
- The referenced expression is an array section with a vector subscript.
- A pointer variable is referenced.
- 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 statement or construct."

-*[*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.

-COMMENT *comment*

Insert comment into executable file

Link only. Default: no comment

Specify `-comment` to insert a comment line into an executable file. If *comment* contains space or tab characters, it must be enclosed in double quotes.

-*[*N*]*CONCC

Support carriage control characters in console I/O

Compile only. Default: -concc

Specify -nconcc to turn off Fortran carriage control processing for console I/O.

-*[*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 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 64 bit double-precision. If you use `-dbl`, all source files (including modules) in a program should be compiled with `-dbl`. Specifying `-dbl` may or may not result in a somewhat slower executable.

-*[*N*]*DLL

Dynamic Link Library

Link only. Default: `-ndll`

Specify `-dll` to create a 32-bit Windows dynamic link library (for more information, see *”Dynamically linked applications”* on page 54).

-*[*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 intrinsic type.

-FILE *filename*

Filename

Compile *and* link. Default: not present

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

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 option, `-checksum`, to the linker and assumes the following string, “`bill.f90`”, is an argument to the `-checksum` option.

```
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` and `.f` files

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

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) that mix 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 option 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

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

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

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

-I *path1[;path2 ...]*

Include Path

Compile only. Default: current directory

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, not spaces. If a space appears as part of a pathname, the entire path must be enclosed in quotes.

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`

-*[*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*]*INLINE *[(arg[,arg[,...]])]*

Inline Code

Compile only. Default: -ninline

Specify -inline to cause user-defined procedures to be inserted inline at the point they are referenced in the calling code. This option only affects code which is in the same source file as the calling procedure. Intrinsic functions, module procedures and internal procedures are not inlined.

Multiple arguments are separated by commas. At least one argument must be present.

If *arg* is a number, any user defined procedure with total lines of executable code smaller than arg is inlined. This argument may only appear once in the argument list.

If *arg* is a number with the letter capital “K” appended, arrays which have a size less than *arg* kilobytes are inlined. Inlining arrays can enhance the optimization abilities of the compiler. This argument may only appear once in the argument list.

If *arg* is a procedure name, or comma separated list of procedure names, the named procedures are inlined.

If *arg* is absent, all procedures having fewer than 30 lines of code and all local data are inlined.

Use of the -inline option may cause long compile times, and may lead to very large executables.

-*[*N*]*LI

Lahey Intrinsic Procedures

Compile only. Default: `-li`

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

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

Library Path

Link only. Default: current directory.

The `-LIBPATH` option 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.

Example

```
LF95 main.obj -libpath d:\mylibs -lib mine.lib
```

-*[*N*]*LONG

Long Integers

Compile only. Default: `-nlong`

Specify `-long` to extend all default INTEGER variables, arrays, constants, and functions to 64 bit INTEGER. If you use `-long`, all source files (including modules) in a program should be compiled with `-long`.

-*[*N*]*LST *[***(f=***fval[***,i=***ival]***)***]*

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 compile session. By default, listing file names consist of the root of the first source file name plus the extension .lst.

You may optionally specify `f` for the listing file name, or `i` to list the contents of INCLUDE files.

fval 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.

ival 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 [(f=fval[,i=ival])]"

-*[*NO*]*MAP *filename*

Change map file name

Link only. Default: create a map file with same name as output file

The `-MAP` option is used to specify a name for the linker map file. The linker map file is a text file describing the output load image. The map file contains the following information:

- 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` option renames or relocates the map file. The option takes a single argument, which is the path and name of the map file to be produced. 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` option. The option takes no arguments. The `-NOMAP` option is useful to make the linker run faster, since no time is spent writing a map file. The option is also a good way to save disk space, because map files can be quite large.

Examples

```
LF95 moe.obj larry.obj curly.obj -map stooges.nuk
LF95 hello.obj -nomap
```

-*[*N*]*MAXFATALS *number*

Maximum Number of Fatal Errors

Compile only. Default: `-maxfatals 50`

Specify `-maxfatals` to limit the number of fatal errors LF95 will generate before aborting. If no argument is specified, the driver will abort with an error message.

If `-nmaxfatals` is specified, no argument is allowed.

-ML { bc | bd | fc | lf90 | lf95 | msvb | msvc | winapi }

Mixed Language

Compile *and* Link. Default: `-ml lf95`

Specify the `-ml` option if your code calls or is called by code written in another language or if your code will call procedures in DLLs created by LF95. `-ml` affects name mangling for procedure names in DLL_IMPORT, DLL_EXPORT, and ML_EXTERNAL statements. See *”Mixed Language Programming”* on page 53 for more information.

Use `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 { bc | bd | fc | lf90 | lf95 | msvb | msvc | winapi }

Mixed Language Default

Compile only. Default: `-mldefault lf95`

Specify the -mldefault option to set the default target language name decoration/calling convention for all program units. Use the -ml option to alternatively affect name mangling only for procedure names in DLL_IMPORT, DLL_EXPORT, and ML_EXTERNAL statements.

Use `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 *dir1[;dir2 ...]*

Module Path

Compile only. Default: current directory

Specify `-mod` *dir* to instruct the compiler to search the specified directory for previously compiled LF95 module files (`.MOD`). If source code containing a module is being compiled, the module `.MOD` and `.OBJ` files will be placed in the first directory specified by *dir*.

When a program that uses a module is linked, the module's object file (or library name) must be provided on the command line. See *”Linking Fortran 95 Modules”* on page 49 for more information and examples.

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`.

-NOLOGO

Linker Banner

Link only. Default: show linker logo

Suppress the LINK version and copyright message.

{ -O0 | -O1 | -O2 }

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 35

Specify `-o1` to perform optimization of object code.

Specify -o2 to perform additional optimizations. This optimization level implements full unrolling of nested loops, loop splitting to promote loop exchange, and array optimizations. Use of the -o2 option may significantly impact compilation speed. Use the -unroll option to limit the level of loop unrolling.

-O *filename*

Object Filename

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

Specify `-o` *filename* 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.

If `-out` is not specified, the output file is not automatically placed in the current directory. By default it is placed in the same directory as the first source or object file listed on the command line.

This option takes a single argument, which is the path and name of the output file. If *filename* contains no path information, the output file is placed in the current directory.

If the file extension `.EXE` is specified, an executable file will be created. If no extension is specified with the `-ndll` option (default), the `.exe` extension is assumed.

If the file extension `.dll` is specified, a dynamic-link library will be created. If no extension is specified with the `-dll` option, the `.dll` extension is assumed.

If the file extension `.LIB` is specified, and the specified library file does not exist, it will be created. If the specified library already exists, it will be updated.

Examples

```
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 option can be used to keep a console window from vanishing at program completion, allowing the user to view the final console output. If `-npause` is specified, the 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 and -WINCONSOLE

-*[*N*]*PCA

Protect Constant Arguments

Compile only. Default: `-npca`

Specify `-pca` to prevent invoked subprograms from storing into constants. The -pca option will silently protect constant arguments and does not produce any warnings.

If -pca is specified in conjunction with -chk (a), the action of -chk (a) is overridden by the action of -pca. In this case, no error is generated when a dummy argument that is associated with a constant actual argument is assigned a new value in the subprogram.

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*]*PREFETCH *[{* **1** | **2** *}]*

Generate prefetch optimizations

Compile only. Default: `-nprefetch`

Prefetch optimizations can improve performance on systems which support prefetch instructions, such as Pentium III and Athlon systems.

The prefetch 1 option causes prefetch instructions to be generated for arrays in loops. The prefetch 2 option generates optimized prefetch instructions. Because Pentium 4 chips implement prefetch in hardware, the use of `-prefetch` can adversely affect performance on those systems. Performance will be program dependent. Try each prefetch option (`-nprefetch`, `-prefetch 1`, or `-prefetch 2`) to determine which works best with your code. The `-prefetch` option will be ignored if `-o0` or `-g` are used.

Please note: code generated with -prefetch is not compatible with processors made before the Pentium III or Athlon.

-*[*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).

-*[*N*]*QUAD

Quadruple Precision

Compile only. Default: `-nquad`

Specify `-quad` to extend all double-precision REAL and double-precision COMPLEX variables, arrays, constants, and functions to 128 bit REAL and COMPLEX respectively. Specifying `-quad` forces `-dbl`, so using `-quad` causes the precision of all REAL variables to be doubled.

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 are trapped by default. This behavior can be overridden using the NDPEXC subroutine or the ERRSET service subroutine (see the file ReadMe_Service_Routines.txt).

-*[*N*]*SAV

SAVE Local Variables

Compile only. Default: `-nsav`

Specify `-sav` to save local variables in between subprogram invocations. `-nsav` causes local variables to be stored on the stack, and their value is not retained in between subprogram invocations. `-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 procedures. Specifying `-nsav` may sometimes require more stack space than provided by default (see *”-STACK reserve[:commit]”* on page 42).

-*[n]*SSE2

Optimize using streaming SIMD extensions

Compile only. Default: `-nsse2`

Specify -sse2 to optimize code using the streaming SIMD (Single Instruction Multiple Data)extensions. This option may only be specified if -tp4 is also specified.

-STACK *reserve[:commit]*

Stack Size

Link only. Default: `-stack 1000000h`

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

reserve is the maximum size of the stack
commit is the increment used when increasing the stack size during runtime

A space must appear between the option and *reserve*.

If a stack segment is already present in the program, then the `-STACK` option changes the size of the existing segment. The linker 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.

LF95 does not allocate local variables on the stack except in these cases:

- Procedures with RECURSIVE keyword
- Procedures with the AUTOMATIC statement/attribute

The LF95 compiler does not have a compiler option to output the required stack size.

A program will not necessarily allocate the maximum amount of stack at the time it is loaded into memory. If it needs more stack during execution, it will dynamically increase the stack.

If your program exceeds the maximum amount of stack at runtime, increase the stack size with `-STACK`. Note that some recursive procedures and files with large arrays compiled with `-nsav` can use very large amounts of stack.

Examples

```
LF95 hello.obj -stack 2000000
LF95 howdy.obj -stack 2000000:10000
```

-*[*N*]*STATICLIB

Static or Dynamic Linking of Fortran Runtime Libraries

Link only. Default: `-staticlib`

Specify `-nstaticlib` to dynamically link an executable or DLL with the Fortran runtime libraries in DLL form.

Specify `-staticlib` to statically link the Fortran runtime libraries with your executable or DLL.

-*[*N*]*STATICLINK

Static Link

Compile only. Default: `-nstaticlink`

Specify `-staticlink` with `-win` and `-ml` to link statically with code produced by another supported language system. See *”Statically linked Fortran and C applications”* on page 58 for more information.

-*[*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 *msgno*

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`.

{ -TP | -TPP | -TP4 }

Target Processor

Compile only. Default: set on installation

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.

Specify -tp4 to generate code optimized for the Intel Pentium 4 processors.

Please note:

Code generated with -tp4 is *not* compatible with processors made previous to the Pentium 4.

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 *and* Link. Default: `-trace`

The `-trace` option causes a call traceback with procedure names and line numbers to be generated with runtime error messages.

-*[*N*]*TRAP *[***d***][***i***][***o***][***u***]*

Trap NDP Exceptions

Compile only. Default: `-ntrap`

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

Table 4: NDP Exceptions

NDP Exception	Option 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. At least one argument must be specified when the `-trap` option is used.

Note that the zero divide exception for two and four byte integers is always handled by hardware, and is not affected by the trap option.

Note that trapping cannot be disabled when `-quad` is specified, except by using the NDPEXC subroutine or the ERRSET service subroutine (see the file ReadMe_Service_Routines.txt).

-*[*N*]*UNROLL *[(limit)]*

Compile only. Default: `-unroll`

Loop unrolling

Specify -unroll (*limit*) to control the level of loop unrolling.

limit is a number in the range 2 ≤ *limit* ≤ 100 enclosed with parenthesis, and denotes the maximum level of loop expansion.

If *limit* is omitted, the value of *limit* is determined by the compiler.

Note that -O0 causes -nunroll to be sent to the compiler by default, but this can be overridden by specifying -O0 -unroll.

-*[*N*]*VARHEAP *[(size)]*

Compile only. Default: `-nvarheap`

Place local variables on heap

Specify -varheap to cause local variables to be allocated on the heap rather than on the stack.

size is a number greater than 4095 enclosed in parenthesis. It is the minimum variable size in bytes that will be placed on the heap. Variables smaller than *size* bytes are not placed on the heap.

If *size* is omitted, it defaults to 4096.

Use the -varheap option when creating programs that have large local arrays. If you notice that increasing the size of a local array causes a stack overflow, using -varheap may alleviate this condition.

Note that the -varheap option does not apply to variables having the SAVE attribute, which includes initialized variables.

-VERSION

Display Version, Copyright and Registration Information

Disables compile and link. Default: `none`

The `-version` option causes the compiler version, copyright and registration information to be printed. Any other options specified on the command line are ignored.

-*[*N*]*W

Compiler Warnings

Compile only. Default: `-w`

Specify `-nw` to suppress compiler warning and informational messages.

-WARN, -FULLWARN

Linker Warnings

Link only. Default: no warnings

The linker detects conditions that can potentially cause run-time problems but are not necessarily errors. LF95 supports two warning levels: `-warn`, and `-fullwarn`.

`-warn` enables basic linker warning messages.

`-fullwarn` provides the maximum level of warning and informational messages.

-*[*N*]*WIDE

Extend width of fixed source code

Compile only. Default: `-nwide`

Using the -wide option causes the compiler to accept fixed form source code out to column 255. The default is to accept code out to column 72

-WIN or -WINCONSOLE

Windows

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

Specifying `-winconsole` will create a console mode application. A Windows console will be created if the program is invoked from Windows Explorer, a menu selection, or a program icon, and it will disappear after program completion unless the `-pause` option is specified. If the program is invoked from the command line of an existing console, all console I/O will be performed within that console.

Specifying `-win` will create a Windows mode application. Under Windows 9x, console I/O is not permitted if the `-win` option was specified. Console I/O with `-win` is allowed if your program is running under a newer Windows operating system. If your program reads from or writes to standard output, a console will be created and will disappear upon program completion.

See also
-*[*N*]*PAUSE

-*[*N*]*WO

Warn Obsolescent
Compile only. Default: `-nwo`

Specify `-wo` to generate warning messages when the compiler encounters obsolescent Fortran 95 features.

-*[*N*]*XREF *[*`(f`=*fval[*`,i`=*ival]*`)`*]*

Cross-Reference Listing
Compile only. Default: `-nxref`

Specify -xref to generate cross-reference information. This information is shown in the listing file in addition to the information that the -lst option would provide. Note that -xref overrides -lst. By default, cross reference file names consist of the root of the source file name plus the extension `.lst`.

You may optionally specify `f` for the listing file name, or `i` to list the contents of INCLUDE files.

fval 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.

ival 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 -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

Include Variables Initialized to Zero

Compile only. Default: `-zero`

Specifying `-zero` will cause all variables which have been explicitly initialized to zero to be given initialization values in the object file.

Specifying `-nzero` will cause explicit initializations to zero to not be given initialization values in the object file, but to be initialized at load time. This will cause object files created with `-nzero` to potentially be much smaller.

Note that specifying the -CHK (u) option will diagnose undefined variables that are not explicitly initialized or assigned by your Fortran code, even when `-zero` is specified.

-*[*N*]*ZFM

Enable zero flush mode for SSE2 instructions

Compile only. Default: `-nzfm`

Specify -zfm to enable zero flush mode for SSE2 instructions. This option may only be specified if -sse2 and -tp4 are also specified.

Note that using -zfm will disable trapping for floating underflow. If an underflow condition occurs during execution of an SSE2 instruction, the affected variable is set to zero. If this behavior presents a problem, use the -nzfm option to guarantee that the underflow exception is thrown.

Linking

Linking an application should always be done under the control of the LF95 driver; it is not necessary to separately invoke `LINK.EXE.` Linking will occur automatically, as long as the `-c` option is not specified. Any combination of source, object, library, and resource files can be provided on the LF95 command line, and LF95 will first compile files using the Fortran or resource compiler, depending on the extension of the specified file, and then invoke the linker.

During the link phase, the driver submits object files, object file libraries, and compiled resource files to the linker. The linker reads individual object files and 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.

Link Environment Variables

LINK

The linker will examine the LINK environment variable, and will use any linker options it finds. If any conflicting options are specified on the command line, they will override those in the environment variable.

LIB

The linker will examine the LIB environment variable, and will use any path information it finds when searching for libraries or object files.

Additional Linker Options

In addition to the link options described in the above section, Microsoft-specific link options which are not documented here may be used on the LF95 command line. These options should be specified with a hyphen (-), not a slash (/) and are sent to the linker unmodified. The linker is fully documented on Microsoft's web site at:

http://msdn.microsoft.com

Linking Fortran 95 Modules

An object file that is created when compiling a module is treated like any other object file. When linking a program that uses a module, the module's object file must be provided to the linker along with the rest of the program. This can be done in one of several ways:

- If the module was already compiled, the object file can be provided along with the other filenames that comprise the program at the time the program is linked.
- If several modules are being used, their object files may be placed in a static library, and the library name can be supplied when linking.
- The module source can be compiled and linked at the same time as the other source files that make up the program. This can be done by specifying all the source that makes up the program on the LF95 command line without specifying the -c option. If this is done, the module source files should appear on the command line before any source files that use the module, and the executable name should be specified using the -out option.

Object File Processing Rules

Object files are processed in the order in which they appear on the command line. If an object file has no path information, it is searched for first in the current directory, and then in any directories listed in the LIB environment variable.

Linking Libraries

No special switch is needed to indicate that a library is to be linked, the driver is able to infer that the file is a library due to the .lib extension. A library should be specified by listing the library file name with the .lib extension on the command line in the same way that a source or object file is specified.

Libraries are searched in the order in which they appear on the LF95 command line. If multiple libraries contain the same object, the first object encountered is linked, and any duplicate objects in subsequent libraries are ignored.

If a library file is specified without path information, the linker looks for it in the following order:

1. In the current working directory
2. In any directories specified with the `-LIBPATH` option.
3. In any directories specified in the LIB environment variable.

Recommended Option Settings

If an `lf95.fig` file exists in the current directory, examine its contents to insure that it contains the desired options.

For debugging, the following option settings will provide an increased level of diagnostic ability, both at compile time, and during execution:

```
-chk -g -trace -info
```

The `-pca` option may be additionally be used to check for corruption of constant arguments. If the results are correct with `-pca` but bad with `-npca` a constant argument has been corrupted.

For further analysis during development, consider specifying any of the following options:

```
-ap -chkglobal -f95 -lst -sav -wo -xref
```

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

For production code, we recommend the following option settings:

```
-nap -nchk -nchkglobal -ndal -ng -o1 -npca -nsav -nstchk
-ntrace
```

Use `-tp, -tpp, or -tp4` depending on your preferred target processor.

Note

- Use of `-tpp` will require that the program be run on a Pentium pro processor or later.
- Use of `-tp4` will require that the program be run on a Pentium 4 processor or later.

For additional optimization, experiment with the `-nprefetch, -prefetch 1` or `-prefetch 2` options and select the one which provides the best performance.

If the program performs many I/O operations, consider tuning the blocksize with the -block option.

Programs may be tuned with the -o2 and the -inline option to increase optimization and to inline code and data.

If the target processor is a Pentium III or Athlon, consider experimenting with the `-nprefetch, -prefetch 1` or `-prefetch 2` options to select the one which provides the best performance.

If the target processor is a Pentium 4, consider tuning with the `-sse2` and `-zfm` options.

If optimization produces radically different results or causes runtime errors, try compiling with `-info` to see exactly which steps are being taken to optimize. The `-info` option also generates warnings on sections of code that are unstable and therefore may cause problems when optimized. A common example of such code is an IF statement that compares floating-point variables for equality. When optimization seems to alter the results, try using the `-ap` option to preserve arithmetic precision while still retaining some optimization.

3 Mixed Language Programming

Mixed language programming is the process of melding code created by different programming languages into an executable program. There are two possible ways that this might be accomplished: by creating object files with different compilers that are linked into a single executable (static linking); or by creating a dynamic link library with one language, and calling procedures from the library using the other language (dynamic linking). Static linking mixes the different language parts into a single executable program which is self contained. Dynamic linking keeps the different language parts separate, and results in two separate entities, a DLL created with one language, and an executable created with the other language.

Regardless of the method chosen to create a mixed language application, two basic problems need to be overcome by the programmer in order to be successful:

- The first problem involves how each language system names its procedures, and how names from one language system can be recognized by the other language system. Each procedure needs to know how the other is named, so that each can call and be called by the other within the execution environment. If the translation between the different naming conventions is not properly done, the programmer will not be able to link the different program parts together, because linker errors concerning unresolved symbols will occur. Resolving the naming problem involves declaring any Fortran procedure names that are to be called from another language, declaring the other language procedure names that will be called in Fortran, and telling LF95 what calling convention is being used at compile time with the `-ml` compiler option. If a DLL is being used, a "translation" between the exported DLL procedures and how Fortran declares the procedures is provided in the form of an import library.

  LF95 code that calls or is called by another language makes the name available by giving it the `DLL_IMPORT`, `DLL_EXPORT` or `ML_EXTERNAL` attribute. The `DLL_IMPORT` attribute is used when calling a procedure from a DLL. The `DLL_EXPORT` attribute is used to make a procedure name externally available when creating a Fortran DLL. The `ML_EXTERNAL` attribute is used to make a procedure from another language available to Fortran or making a Fortran procedure available to be called from another language when static linking. At compilation time, any procedure names having one of these attributes are 'decorated' to match the calling convention specified by the `-ML` option.

- Secondly, in order to be useful, the procedures need to be able to pass information back and forth in a way that both can understand and utilize. This involves the passing of arguments to a subroutine or function, passing a function result between language systems, and how basic data types are interpreted by each language system. If arguments are not passed or interpreted correctly, the result can be unpredictable, and can range from nonsense answers to the program crashing with an "illegal operation" message. The arguments passing problem is addressed for each supported language system, described in subsequent sections.

Dynamically linked applications

A dynamically linked application consists of two parts: a separately created dynamic link library (DLL), and an executable program which references the DLL. A DLL is a collection of subprograms 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.

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. Note that issuing a STOP statement from within a Fortran DLL will cause the entire program to terminate. If you are calling DLL procedures from a language system other than LF95, please refer to that language system's DLL documentation for more information.

Supported language systems

Lahey/Fujitsu Fortran 95 supports DLL calling conventions for the following languages systems:

Table 5: Compiler Support for Lahey DLLs

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

Declaring calling conventions

In order to reference a procedure across a DLL interface, the LF95 compiler must be informed of the procedure name to be exported, and given a calling convention for the external names in your DLL. The procedure names that will be externally available 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 procedure names appearing in a `DLL_EXPORT` or `DLL_IMPORT` statement are *case sensitive* (unlike the Fortran naming convention, which ignores case). `DLL_EXPORT` is used to define an externally available DLL procedure, and `DLL_IMPORT` is used when referencing a DLL procedure. The calling convention is defined with the use of the `-ML` compiler option. 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.

Table 6: -ML Options

Option	Compiler
`-ml lf95`	Lahey/Fujitsu Fortran 95
`-ml lf90`	Lahey Fortran 90
`-ml bc`	Borland C++
`-ml bd`	Borland Delphi
`-ml msvc`	Microsoft Visual C++
`-ml msvb`	Microsoft Visual Basic
`-ml winapi`	Windows API functions invoked directly from Fortran

LF95 can build DLLs callable from Microsoft Visual Basic, however, Microsoft Visual Basic does not build DLLs callable by LF95. Assembly procedures may be called from Fortran procedures, however the use of DOS interrupts is not supported.

Building Fortran DLLs

When you create a Fortran DLL, you must indicate the procedures that you want to export from the DLL with the `DLL_EXPORT` attribute. 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., `Foo` is the same as `FOO`. Note that the compiler allows you to build your DLL from multiple `.OBJ` files.

Example code

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

The code must be compiled using one of the options shown in Table 6, “-ML Options,” on page 56. When the `-dll` option specified, a DLL is created and a Microsoft-compatible import library is generated.

Example build command

```
lf95 -dll -win -ml msvc half.f90
```

The above command creates the files `half.dll` and `half.lib` which are compatible with Microsoft Visual C.

Building Import Libraries

A Microsoft-compatible import library is automatically generated whenever LF95 is used to create a DLL. When an LF95 program that calls a DLL is linked, a Microsoft-compatible import library must be provided. Usually, the vendor that supplies the DLL will also provide a Microsoft-compatible import library. Additional information on import libraries can be found in Chapter 5, *Library Manager* under the heading *"Creating import libraries"* on page 112.

Building import libraries from object files

If the user is building a non Fortran DLL on site for use with LF95, and a Microsoft-compatible import library is not created, an import library can be generated from the object files using `LIB.EXE`. Doing this entails making a definition file which contains the names of the exported procedures, and running `LIB` with the following command:

`LIB /def:`*defile.def file1.obj file2.obj* `/out:`*implib.lib*

Where:

defile.def is the name of the definition file. Lahey provides a utility *"MAKEDEF.EXE"* to generate definition files given a DLL. Alternatively, the `DUMPBIN` utility can be used to list exported symbols from the DLL; then the definition file can be created by hand. Note that any export that appears in the definition file must be present in the object file, otherwise an unresolved reference will occur when the `LIB` command is executed. If this happens, it is usually sufficient to remove the unresolved reference from the definition file.

file1.obj and *file2.obj* are object files that were used to build the DLL.

implib.lib is the name of the import library to be produced.

Building import libraries when no object file is available

Occasionally, the situation occurs when only a DLL is available, without an import library or object files. If the user knows how to call the DLL procedure from Fortran, an import library can be generated using a stub program. A stub program is a skeleton that contains function or subroutine statements including any argument calling sequences, argument declarations, a `DLL_EXPORT` statement, and end statement, but no other source code - much like a procedure would appear inside an interface block. The stub file is compiled to create an LF95 object file, using an appropriate `-ml` option, and `-c.` Once the stub object file is created, the import library can be generated using the instructions in the preceding section: *"Building import libraries from object files"*. During execution of the `LIB` command, a warning concerning duplicate symbols may appear, if a non Microsoft convention is used, but it can be disregarded. Note that `-ml lf95` should never be used to create import libraries from stubs. Code that calls the DLL should be compiled using the same `-ml` option that was used to compile the stub file. Note that the definition file that is used when creating the import library should only contain procedure names that appear in the stub file, otherwise unresolved references will occur when the LIB command is executed. An example of creating import libraries using stubs appears in the EXAMPLES\MIX_LANG\BD directory.

Building non Microsoft import libraries for Fortran DLLs

If the user wishes to build an import library for a Fortran DLL that is called from a language that does not accept Microsoft-compatible import libraries, the 3rd party language will usually provide a method of generating a compatible import library, such as Borland's IMPLIB.EXE. In some cases, the 3rd party linker may provide commands that enable DLL references to be resolved. Consult the documentation provided with the 3rd party compiler and tools for instructions on resolving references to DLL procedures.

Examples of how to build Fortran callable DLLs from non Fortran languages, and how to generate Microsoft compatible import libraries from non Microsoft object files reside in directories under the EXAMPLES\MIX_LANG directory.

Delivering Applications with LF95 DLLs

When you deliver applications built with LF95 DLLs, you must include the DLLs and associated import libraries you created. At runtime, all of the DLLs must be available on the path or in a directory that Windows checks for DLLs.

Statically linked Fortran and C applications

Statically linked applications consist of 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++ and Fujitsu C (FCC). LF95 is also static link compatible with object code created with Borland C/C++, but at this time it is not possible to reliably call C runtime procedures from Borland C.

Much of the following information is provided courtesy of Kenneth G. Hamilton --- 7-Oct-1998.

There are several reasons why you may wish to call a C function from your Fortran code. For example, you may have the C source code for a function that you need, but not want to take the time to recode it into Fortran. Or, you may wish to take advantage of some feature of C, such as unsigned integers, that is not available in Fortran. Additionally, many current operating systems, including Microsoft Windows, are written in C and the authors have not seen fit to document the interface to the system services in any other language.

You should, however, keep in mind that as a consequence of the extensive use of pointer arithmetic, C code ultimately cannot be optimized as well as Fortran. Most examples of fast, efficient, C code are the result of a great deal of programmer labor, just as is the case in assembly language coding.

Building statically linked applications

The information on building a statically linked program is the same as for dynamic linking (described above) with the following exceptions:

- Specify the `-staticlink` and `-ml` options on the LF95 command line (do not specify `-dll`).
- Use `ML_EXTERNAL` instead of `DLL_IMPORT` or `DLL_EXPORT` in your Fortran source match the calling conventions of Visual C++ at compile time. If using Fujitsu C, calling conventions can be matched by following the instructions in the section *”Calling Conventions”* on page 59.
- You must have a Fortran main program.
- Import libraries are not included on the LF95 command line (import libraries are specific to DLLs).
- Fortran common blocks are aligned on one-byte boundaries. To align your C structures along one-byte boundaries, use the `/Zp1` option or the `pack` pragma with Microsoft Visual C++. Use the `-a-` option or the `option -a-` pragma with Borland C++. Note that use of these options 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.

There are several examples in the following sections. The source code, to enable you to experience mixed-language calling, are in subdirectories `examples\mix_lang\fcc\ex1, ex2, ex3,` etc., below your main LF95 directory. Each one is accompanied by a file called GEN.BAT, that will compile and link the sample code. There are additional examples specific to compiler type in the `examples\mix_lang\msvc` and `examples\mix_lang\bc` directories.

Calling Conventions

When it compiles Fortran source and emits object code, LF95 converts the names of all entry points and external references into all lower case letters, and attaches an underscore (_) symbol to both the front and back of each such name. FCC does not change the case of names, but it does add a leading underscore to each one.

Therefore, if a Fortran program calls a subroutine named "CLOUD", LF95 will generate a requirement for an external symbol called "_cloud_". If the subroutine is written in C, and compiled by FCC, then the entry point name must be "cloud_". (Note the absence of a leading underscore, which will be added by FCC.)

Argument Passing

Traditionally, Fortran compilers arrange for arguments to subroutines to be passed by reference. This means that the address is passed (pushed on the stack, in the case of PCs) and so the called routine has full access to the variable in the caller, and can read or write to that location.

C compilers, on the other hand, pass simple (i.e., scalar) variables by value, meaning that the current value of that variable is pushed, rather than its address. The function that is called can thus read, but cannot change, the variable in the caller. More complicated objects, such as arrays and structures, are passed by reference by C compilers. (Confusion over which symbols represent values, and which addresses, is a common source of bugs in C programs, and so you should check your usage carefully.)

Trying to connect a Fortran caller to a C callee thus requires that one bridge these two conventions. It is possible to do this either by modifying the Fortran part or the C portion of the calling interface. Since LF95 is a Fortran package, in the examples that follow we will leave the Fortran form alone and modify the C side. This essentially means that C functions should be set up so as to expect that all visible arguments are being passed by reference, or "as pointers" in the C lingo.

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 automatic arrays.

Variable Type Correspondence

When passing arguments to a subprogram, it is necessary that they match the list of formal parameters on the entry point. The following table shows what various Fortran variable types correspond to in C.

Table 7: Variable Type Equivalents

Fortran Type	Kind No.	C Type
INTEGER	1	char
INTEGER	2	int
INTEGER	4	long
REAL	4	float
REAL	8	double
COMPLEX	4	struct{float xr, xi;}
COMPLEX	8	struct{double xr, xi;}
LOGICAL	1	char
LOGICAL	4	long
CHARACTER	1	(none)

The C language allows unsigned integers of various lengths. There is no direct analog of this in Fortran, however the unsigned integers that can be returned from C can be stored and slightly manipulated in Fortran. Fortran cannot perform arithmetic on unsigned integers, however this is often unnecessary: one of the most common uses for unsigned integers is as handles for files, windows, and other objects.

Handles are received, copied, and passed into other routines, but are never subjected to computation. It is therefore possible to treat a handle as simply an INTEGER of the appropriate length and there will be no problem. If it is necessary to display the value of a handle it can be done in hexadecimal (Z) format, with no loss of information.

Example 1 --- A Simple Subroutine

First, let us look at the simplest example of a Fortran program calling a C subroutine. The following main program defines two integers, I and J, and then calls SUB to add them and return the sum.

```
PROGRAM MAIN
integer :: i,j,k
i = 12
j = 43
k = 0
print *, 'Before: i,j,k=',i,j,k
call sub(i,j,k)
print *, 'After:  i,j,k=',i,j,k
stop
end
```

This is the subroutine that performs the addition.

```
void sub_(i,j,k)
int *i, *j, *k;
{
  *k = *i + *j;
  return;
}
```

In C, a subroutine is a function of type "void." As we noted earlier, the name of the subroutine must be in lower case letters, with a trailing underscore. Since Fortran normally passes arguments by reference, the C subroutine must receive them as pointers (hence the "*" in front of the variable names). The type INTEGER variables in Fortran are treated as type "int" in C.

Example 2 --- Passing Real Arguments

The situation is the same when floaing point arguments are passed. In this example, three default REAL(KIND=4) arguments are sent to a C subroutine, where they are manipulated.

```
PROGRAM FLTMAIN
x = 2.17
y = 5.6
z = 0.0
print *,' x,y,z=',x,y,z
call cmult(x,y,z)
print *,' x,y,z=',x,y,z
stop
end
```

This is the C subroutine, where the REAL(KIND=4) variables are received as pointers to variables of type "float." If the arguments were REAL(KIND=8), then the C side would expect them as type "double."

```
void cmult_(x,y,z)
float *x, *y, *z;
{
  *z = *x * *y + 2.0;
  return;
}
```

Example 3 --- Passing CHARACTER Arguments

Passing type CHARACTER variables poses a higher level of complexity. Consider the following main program, which assigns a literal string to A, and then calls CHRCOPY to duplicate A into B.

```
PROGRAM CHRMAIN
character*20 a, b
a = 'This is a message'
b = ' '
print *, 'a=',a
print *, 'b=',b
call chrcopy(a,b)
print *, 'a=',a
print *, 'b=',b
stop
end
```

When LF95 passes type CHARACTER arguments to a subroutine, it actually sends both the starting address of the string, plus the length (which is passed by value). The lengths of any CHARACTER arguments are treated as hidden arguments to the right of the normal argument list.

Thus, in the following C subroutine, the argument list consists of four items, even thought we could see only two in the Fortran CALL statement. The first two arguments are the character strings, passed by reference so that they appear here as pointers to variables of type "char." (Type "char" in C is not a true string variable, but is rather a one-byte integer.)

The third and fourth arguments are the lengths of A and B, passed by value. We can tell that they are being passed by value here because they are not prefixed by asterisks, but just appear as plain variables.

```
#include <string.h>
void chrcopy_(a,b,na,nb)
char *a, *b;
int na, nb;
{
  int nmin;
  nmin = na > nb ? nb : na;
  strncpy(b,a,nmin);
  return;
}
```

The subroutine first compares the lengths of the two CHARACTER variables, and then selects the minimum (in case they are different). That becomes the number of characters to copy from A to B, and the C library routine "strncpy" is used.

Example 4 --- Passing ASCIIZ Arguments

In early Fortran compilers, character strings were stored as arrays of numeric storage locations, packed several characters to each word and then terminated by a word or partial word of zero. Because different types of computer have different word lengths, this "Hollerith" scheme often led to seriously non-transportable code. Some computers stored four characters per word, while others stored five, six, eight, or ten characters per word and so many routines that performed input or output required drastic reworking when moved from one brand of computer to another.

When the Basic language was released in the 1970s, it introduced the notion of a special "string" data type that was independent of the hardware design. This was such a good idea that it was copied into the 1977 Fortran standard as CHARACTER variables.

Unfortunately, at the same time that Fortran was copying from Basic, C was copying from Fortran and so currently C compilers still expect character strings to be stored as an array of numeric storage locations (usually bytes), terminated by a null. In some cases, you may find it preferable to pass CHARACTER variables to C by appending a null, so that it looks like the legacy method expected by the C language. In order to do this, you would change

```
CALL CSUB(...,ASTR,...)
```

into

```
CALL CSUB(...,ASTR//CHAR(0),...)
```

where ASTR is a CHARACTER variable. In this case, however, the Fortran compiler will make a copy of ASTR with the null attached, and pass that. This means that the subroutine will not be able to modify the original string since ASTR//CHAR(0) is an expression rather than a variable, but that may well be desirable.

If you want to allow the subroutine to modify the original string, then you should add the null into the CHARACTER variable, as shown in the following example.

```
PROGRAM CHRMAIN
character*20 a, b
a = 'Original text'//char(0)
b = ' '
print *, 'a=',a
print *, 'b=',b
call chrcaps(a,b)
print *, 'a=',a
print *, 'b=',b
stop
end
```

Here is a C subroutine that returns B as a capitalized version of A, as required by the main program.

```
void chrcaps_(a,b,na,nb)
char *a, *b;
int na, nb;
{
  char *i, *j;
  for (i = a, j = b; *i != 0; i++, j++) {
    *j = *i;
    if (*j >= 97 && *j <= 122) *j -= 32;
    }
  return;
}
```

In this case, the copying operation is halted by the appearance of a null (the "*i != 0" clause in the "for" statement). Local pointer variables *i and *j are used instead of the ones that were supplied by the caller.

Example 5 --- Accessing COMMON Blocks

When LF95 processes COMMON blocks, it modifies them in the same way as it does entry points. That is to say that a block named /SAND/ will invisibly become a global object named "_sand_" and this alteration must be dealt with when performing inter-language calling. The secret name of blank COMMON is "__BLNK__", with two underscores in front and behind.

Here is an example of a Fortran main program that supplies values to some variables that are in COMMON blocks, one blank and one named.

```
PROGRAM CMN_MAIN
integer :: i
real :: x,y,z
common /zulu/ x, y
common z
i = 12
x = 4.5
y = 0.0
z = 8.1
print *, 'Before: i,x,y,z=',i,x,y,z
call ccmn(i)
print *, 'After: i,x,y,z=',i,x,y,z
stop
end
```

That program calls the following C subroutine:

```
extern struct
{
  float x, y;
} zulu_;

extern struct
{
  float z;
} _BLNK__;

void ccmn_(i)
int *i;
{
  zulu_.y = zulu_.x + (float)(*i);
  _BLNK__.z += zulu_.x;
  return;
}
```

In order to access the COMMON blocks from C, we must define a pair of structures, and declare them outside of the function body so that they acquire the global attribute and can connect to the COMMON blocks that the Fortran compiler is going to set up.

Since C prepends an underscore to the global names, the named common /ZULU/, which is called "_zulu_" in the object modules, must be called "zulu_" (no leading underscore) in the C code. Likewise, the blank COMMON, called "__BLNK__" in the object code, is called "_BLNK__" (only one leading underscore) in C.

Example 6 --- Functions

Calling a function that is written in C, one that returns a value (as opposed to a subroutine), is fairly simple as long as you make sure that the type of the function in C matches what Fortran expects to receive.

Here is an example of a Fortran main program that calls several C functions, each of a different type. The argument lists are the same for all the functions: two default integers, but the return value differs.

```
      PROGRAM MAIN
      integer :: i,j
      integer(kind=1) :: k1, i1add
      integer(kind=2) :: k2, i2add
      integer(kind=4) :: k4, i4add
      real(kind=4) :: r4, r4add
      real(kind=4) :: r8, r8add
      external :: i1add, i2add, i4add, r4add, r8add
!
      i = 12
      j = 43
      k1 = 0; k2 = 0; k4 = 0
      print *, 'Before: i,j=',i,j
      k1 = i1add(i,j)
      k2 = i2add(i,j)
      k4 = i4add(i,j)
      print *, 'After:  k1,k2,k4=',k1,k2,k4
      r4 = r4add(i,j)
      r8 = r8add(i,j)
      print *, 'r4,r8=',r4,r8
!
      stop
      end
```

These are the C functions called by the Fortran main. Note that the type of variable for a function to return is specified in the opening statement, in place of the "void" that was used in the earlier subroutines.

```
char i1add_(i,j)
int *i, *j;
{
  char k;
  k = *i + *j;
  return(k);
}

short i2add_(i,j)
int *i, *j;
{
  short k;
  k = *i - *j;
  return(k);
}

long i4add_(i,j)
int *i, *j;
{
  long k;
  k = *i * *j;
  return(k);
}

float r4add_(i,j)
int *i, *j;
{
  float r;
  r = (float)(*i) + (float)(*j);
  return(r);
}
```

```
double r8add_(i,j)
int *i, *j;
{
  double d;
  d = (double)(*i) / (double)(*j);
  return(d);
}
```

Fortran Calling Fortran DLLs

Even though the same language system is used to create both the DLL and the executable, the mixed language rules must be observed. Create the Fortran DLL as described in *”Building Fortran DLLs”* on page 56, building with the `-ml lf95` compile option:

```
lf95 source.f90 -win -dll -ml lf95
```

LF95 builds the DLL `source.dll`. It also generates a `source.lib` file containing definitions needed to link to this DLL.

Next build the Fortran Main with:

```
lf95 main.f90 -win -ml lf95 source.lib
```

To run the program, the DLL must be in the same directory as the executable, or in a directory on the path.

Fortran and C applications

Fortran calling C DLLs

When you create a Fortran procedure that references a C procedure you declare the C procedure name with the `DLL_IMPORT` attribute in your Fortran code. The procedure may be a subroutine or function. C functions may only return the Fortran equivalent of default INTEGER, REAL, or LOGICAL results.

Example code:

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

Before building the Fortran main program with LF95, you must have a DLL and import library available. Refer to your C manual for specifics on creating a DLL. If the C compiler does not create a compatible import library (`.LIB` file) for the DLL, proceed as described above, *"Building Import Libraries"* on page 57.

If the DLL was created with Microsoft Visual C++, use the `-ml msvc` option:

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

If the DLL was created with Borland C++, use the `-ml bc` option:

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

Where *dll_src*`.lib` is the name of the Microsoft compatible import library.

There are examples of calling C DLLs in the directories below LF95's `EXAMPLES/MIX_LANG` directory.

C Calling Fortran DLLs

Create the Fortran DLL as described in *"Building Fortran DLLs"* on page 56, building with the `-ml` compile option that matches your C compiler.

To compile your Fortran source for use with Microsoft Visual C++, issue the command:

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

This command will cause a DLL called `source.dll` to be created, as well as an import library called `source.lib.`

To compile your Fortran source for use with Borland C++, issue the command:

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

The user will need to run Borland's `IMPLIB.EXE` to build the import library compatible with the Borland linker. `IMPLIB` is distributed with the Borland compiler, and is not a part of LF95.

Once you've created the DLL and generated the import library, use the C language system to 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.

Referencing DLL Procedures

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

For example, this Fortran function:

```
function dll1(a, a1, i, i1, l, l1)
   integer, dll_export :: DLL1
   real a, a1(10)
   integer i, i1(10)
   logical l, l1(10)
   ...
end function
```

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 LF95's `EXAMPLES\MIX_LANG\MSVC` directory (for Microsoft Visual C++) or LF95's `EXAMPLES\MIX_LANG\BC` directory (for Borland C++).

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. Arguments may be passed by reference (the default) or by value using either the CARG or VAL function. See *"Argument Passing"* on page 60 for more information.

Microsoft Visual Basic Information

Visual Basic calling Fortran

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
```

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` options.

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 `EXAMPLES\MIX_LANG\MSVB` directory.

4. Run the demo (`F5`).

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.

Borland Delphi 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
```

Running the Delphi Calling Fortran Demo

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

2. Ensure that the resulting `BDDEMO2.DLL` resides either in the current working directory, or 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 `EXAMPLES\MIX_LANG\BD` directory.

4. Run the demo (`F9`).

Fortran Calling Delphi DLLs

Before building the Fortran main program with LF95, you must have a DLL and import library available. Refer to your Delphi documentation for the specifics on creating a DLL. Because Delphi does not build a `.LIB` file for the DLL, and does not create compatible object files, the stub method must be used to create a Microsoft-compatible import library. See*"Building import libraries when no object file is available"* on page 57. An example of linking a Fortran program to a Delphi DLL appears in the `EXAMPLES\MIX_LANG\BD` directory.

When you create a Fortran procedure that references a Delphi DLL procedure you declare the Delphi procedure name with the `DLL_IMPORT` attribute in your Fortran code. The procedure may be a subroutine or function. Delphi DLL functions may only return the equivalent of default INTEGER, REAL, or LOGICAL results.

Example code:

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

Build the Fortran program using the `-ml bd` option:

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

Where *dll_src*`.lib` is the name of the Microsoft compatible import library created by the stub method.

Running the Fortran Calling Delphi Demo

1. From Delphi, open `F95CALLBD.DPR` in LF95's `EXAMPLES\MIX_LANG\BD` directory.
2. Build the DLL by pressing `Ctrl-F9`.
3. Copy `F95CALLBD.DLL` to LF95's `EXAMPLES\MIX_LANG\BD` directory.
4. Change to LF95's `EXAMPLES\MIX_LANG\BD` directory.
5. Run the batch file `RUNF95CALLBD.BAT`. This batch file compiles the Fortran stub code, generates an import library, and compiles the Fortran main program using the newly created import library.
6. The resulting executable, `F95CALLBD.EXE` is automatically run by the batch file.

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).

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.

Calling Fortran DLL’s from .NET Applications

Programs created using a .NET language dynamically load unmanaged DLL’s at runtime, so the DLL name and characteristics must be specified in the managed code. When creating a native Fortran DLL that can be called by a .NET application, compile and link with one of the following `-ml` options: `winapi`, `msvc`, `lf95`, or `fc`. If a version of LF95 prior to v5.7 is being used, the `-ml winapi` option should not be specified. DLLs built with the `-ml msvb` option can be called from VB.NET applications. DLLs built with the `-ml` options `lf90`, `bc`, or `bd` cannot be called from .NET languages.

For a DLL compiled and linked with `-ml lf95`, or no `-ml` option, the cdecl calling convention is used.

For a DLL compiled and linked with `-ml winapi`, `-ml msvc`, or `-ml fc`, the stdcall2 calling convention is used.

Fortran function results and argument types must be able to map to .NET variable types.

Example code demonstrating calling Fortran DLL’s from .NET languages exist in directories under the `EXAMPLES\MIX_LANG` directory. These directories all contain `.NET` in the directory name.

Calling LF95 DLLs from Microsoft C#

For DLL’s using the cdecl convention, declare the Fortran procedure in the C# code using the following syntax:

> [DllImport("*dll-name.dll*", CallingConvention=CallingConvention.Cdecl)]
> public static extern *return-type procedure-name_* (*argument-list*);

For DLL’s using the stdcall2 convention, declare the Fortran procedure in the C# code using the following syntax:

> [DllImport("*dll-name.dll*")]

public static extern *return-type procedure-name* (*argument-list*);

Where:

dll-name.dll is the pathname\file-name of the unmanaged (Win32) DLL.

return-type is "void" if calling a Fortran subroutine, otherwise the C# equivalent of the Fortran function return type.

procedure-name is the case-sensitive procedure name. If the cdecl convention is used, a trailing underscore must be appended to the procedure name.

argument-list is a managed code variable list with types mapped to Fortran dummy argument types; precede pass-by-reference parameters with "ref".

Calling LF95 DLLs from Microsoft Visual Basic .NET

For DLL’s using the cdecl convention, declare the Fortran procedure in the VB.NET code using the following syntax:

Calling a function:

```
Class ClassName
  <DllImport("dll-name.dll", CallingConvention:=CallingConvention.Cdecl)> _
  Shared Function proc-name_ (arg-list) as return-type
  End Function
End Class
```

Calling a subroutine:

```
Class ClassName
  <DllImport("dll-name.dll", CallingConvention:=CallingConvention.Cdecl)> _
  Shared Sub proc-name_ (arg-list)
  End Sub
End Class
```

For DLLs using the stdcall2 convention, declare the Fortran function in the VB.NET code using the following syntax:

Calling a function:

```
Class ClassName
  <DllImport("dll-name.dll", CallingConvention:=CallingConvention.StdCall)> _
  Shared Function proc-name (arg-list) as return-type
  End Function
End Class
```

Calling a subroutine:

```
Class ClassName
  <DllImport("dll-name.dll", CallingConvention:=CallingConvention.StdCall)> _
  Shared Sub proc-name (arg-list)
  End Sub
End Class
```

For DLLs compiled using the `-ml msvb` option, declare the Fortran function in the VB.NET code using the following syntax:

Calling a function:

```
Class ClassName
  Declare Auto Function proc-name Lib "dll-name.dll" (arg-list) as return-type
  End Function
End Class
```

Calling a subroutine:

```
Class ClassName
  Declare Auto Function proc-name Lib "dll-name.dll" (arg-list)
  End Function
End Class
```

Where:

dll-name.dll is the pathname\file-name of the unmanaged (Win32) DLL.

return-type is the VB.NET equivalent of the Fortran function return type.

proc-name is the case-sensitive procedure name. If the cdecl convention is used, a trailing underscore must be appended to the procedure name.

arg-list is a managed code variable list with types mapped to Fortran dummy argument types; precede pass-by-reference parameters with "ByRef".

Calling LF95 DLLs from Microsoft Visual C++ .NET

For DLLs using the cdecl convention, declare the Fortran procedure in the C++ code using the following syntax:

[DllImport("*dll-name.dll*", CallingConvention=CallingConvention::Cdecl)]
extern "C" *return-type procedure-name_* (*argument-list*);

For DLLs using the stdcall2 convention, declare the Fortran procedure in the C++ code using the following syntax:

[DllImport("dll-file-name.dll"]
extern "C" *return-type procedure-name* (*argument-list*);

Where:

dll-name.dll is the pathname\file-name of the unmanaged (Win32) DLL.

return-type is "void" if calling a Fortran subroutine, otherwise the C++ equivalent of the Fortran function return type.

procedure-name is the case-sensitive procedure name. If the cdecl convention is used, a trailing underscore must be appended to the procedure name.

argument-list is a managed code variable list with types mapped to Fortran dummy argument types; precede pass-by-reference parameters with "ref".

Calling the Windows API

LF95 can directly access functions in the Windows API, with some limitations. You will need to have access to Windows API documentation and some knowledge of Windows Programming in C or C++ to take full advantage of this functionality, since the API is designed for C and C++.

Complete Windows applications can be written with Lahey Fortran 95 without resorting to using another language for the user interface. This might not be the best approach for many people, but examining how to do it can boost one's understanding of the issues, and these issues can crop up even when creating Windows applications using other approaches.

An example of this approach can be found in LF95's EXAMPLES\MIX_LANG\WINAPI directory, in the files WINDEMO.F90, WINDEMO.RC, WINDOWS.F90, and RUNWINDEMO.BAT.

The first step is to compile the file WINDOWS.F90, found in LF95's SRC directory. Then USE the module WINDOWS_H in any procedure that will call the Windows API. WINDOWS.F90 is a Fortran translation of the standard windows header file WINDOWS.H, which contains definitions for various Windows parameters.

Next declare the API function with the DLL_IMPORT attribute in a type statement, for example, if you want to call the API function MessageBox:

```
INTEGER, DLL_IMPORT :: MessageBoxA
```

Names with the DLL_IMPORT declaration are case sensitive. Elsewhere in your Fortran program the names of imported procedures are case insensitive.

Here are some more things to consider:

- Compile your code using the -ml winapi, -win, and -nvsw options.
- When calling Windows API procedures from Fortran you will need to have DLL_IMPORT statements with the names of all of the API procedures you will use. These names are case sensitive and you will need to use the correct case in the DLL_IMPORT statement. Elsewhere in your Fortran program code the case for these procedure names does not matter, though it's a good idea for clarity's sake to retain the case used in the Windows API. A good place for these DLL_IMPORT statements is in the module you create for your parameter declarations.
- If you have a resource file called MYRC.RC, compile it by adding MYRC.RC to the LF95 command line. You need to include WINDOWS.H (supplied with LF95 in the SRC directory) in your resource file. LF95's driver will call RC.EXE (the resource compiler which ships with LF95 and with various other Windows compilers) to create MYRC.RES. This will then be linked with the other objects and libraries you specified on the command line.
- Any new item you create with a #define in your resource file needs to be declared as an INTEGER parameter in your Fortran source so that it is accessible in the scoping unit in which it is referenced. It is cleanest to put all of these parameter declarations in a module.
- Void API functions must be called as subroutines from Fortran and API functions which return values must be called as functions from Fortran.
- Many of the API functions you call will need to have the letter 'A' appended to the function name. This calls the ASCII (rather than the Unicode) version of the function. If the linker gives you an unresolved external message on an API function you think you've declared properly, try appending an 'A' to the name. It is a good bet that API functions that deal with character strings will require the 'A'.
- API function arguments that do not map to Fortran intrinsic types need to be declared in your Fortran program. Declare structure arguments as SEQUENCE derived types. Declare pointers (to anything, including strings) as INTEGERs.
- Whenever you pass a numeric argument use CARG. For example:

```
call PostQuitMessage(carg(0))
```

- Whenever you pass a pointer argument use CARG(POINTER(*argument*)) instead of *argument*. For example:

```
type (WNDCLASS):: wc
result=RegisterClassA(carg(pointer(wc))
```

- Whenever you pass a pointer to CHARACTER, remember that C requires null-terminated strings. CARG will make a copy of a string and null-terminate it for you. However, because a copy is made, the original value cannot be changed by the function you call. For example:

```
result = SendDlgItemMessageA(carg(hwnd),         &
                             carg(IDC_LIST1,     &
                             carg(LB_ADDSTRING), &
                             carg(0),            &
                             carg(string))
```

To pass a string you want the function to change, null-terminate the string manually and then use CARG of the POINTER. Note that you can use CHAR(0) to generate a null. For example:

```
character(len=81) :: mystr ! leave space for trailing null
mystr = trim(mystr(1:80)) // char(0)
call SomeAPIRoutineA(carg(pointer(mystr)))
```

- Wherever on the right-hand side of a C assignment statement you would use the ampersand character to get the address of something, you will need to use POINTER in your Fortran program. For example:

  ```
  wc%lpszClassName = pointer(szClassName)
  ```

  is equivalent to the C:

  ```
  wc.lpszClassName = &szClassName;
  ```

- Callback procedures, where Windows will be calling a Fortran procedure, must not be module procedures or internal procedures.
- To set up a callback procedure, include an interface block defining the callback procedure and declaring it to be ml_external. Then use the POINTER of the procedure name. For example:

  ```
  interface
     integer function WndProc(hwndByValue,    &
                              messageByValue, &
                              wParamByValue,  &
                              lParamByValue)
        ml_external WndProc
        integer :: hwndbyValue, messageByValue, &
                   wParamByValue, lParamByValue
     end function WndProc
  end interface
  type(WNDCLASS):: wc
  wc%lpfnWndProc = offset(WndProc)
  ```

- Arguments to a Fortran callback procedure are values (C passes by value). To make these work in your callback procedure, assign the pointer of these values to local variables. For example:

```
integer function WndProc(hwndByValue,    &
                         messageByValue, &
                         wParamByValue,  &
                         lParamByValue)
   implicit none
   ml_external WndProc
   integer :: hwnd, message, wParam, lParam
   integer :: hwndByValue, messageByValue
   integer :: wParamByValue, lParamByValue
   hwnd = pointer(hwndByValue)
   message = pointer(messageByValue)
   wParam = pointer(wParamByValue)
   lParam = pointer(lParamByValue)
! do not reference the ByValue arguments from here on !
```

- See `windows.f90` in the SRC directory for examples of functions, types, and definitions for use in Windows API programming.

Calling assembly language procedures

The following information is provided courtesy of Kenneth G. Hamilton, 12-Oct-1998.

LF95 Conventions

This section is intended to assist the experienced assembly language programmer in writing subprograms that can be called by LF95-compiled Fortran code. The examples that follow were processed by Microsoft MASM v6.11a, although any recent assembler will likely suffice. In addition to this information, you should also have on hand appropriate documentation for your assembler. The examples in this write-up can be found in subdirectories EXAMPLES\MIX_LANG\ASSEMBLY\EX1, EX2, EX3, etc.

Each sample program can be compiled and linked by using the GEN.BAT file that accompanies it.

Entry Point Name Mangling

When it compiles Fortran source code, LF95 shifts the names of subroutines and functions into lower case letters, and attaches an underscore symbol (_) both before and after each name. As an example, suppose that an LF95 program calls subroutine RAINBOW. If that routine is written in assembly language, then it must have an entry point called _rainbow_ on a PROC or LABEL statement, and that name must declared to be a PUBLIC symbol.

Saved Registers

LF95 requires that subroutines and functions preserve the value of the EBX, ESI, and EDI registers. If any of these registers are used in an assembly-language routine, they can be saved by pushing them upon entry, and then popping them before returning.

Argument Passing

LF95 passes numeric and logical arguments by pushing their addresses onto the stack, from right to left. Each address is a four-byte quantity, so that, upon entry to a subprocedure the first argument's address is located at ESP+4, the second at ESP+8, the third at ESP+12, and so on. (The ESP register itself contains the address that control will return to in the calling routine, upon subprogram termination.)

Generally, the best procedure (and this is what LF95 itself does) is to push EBP onto the stack, and then move the contents of the ESP register into EBP. This is often known as the ‘preamble’ of the routine. The arguments can then be accessed using EBP instead of ESP, and any additional pushing or popping will not result in any confusion about where the argument addresses are. Since pushing EBP onto the stack changes the stack pointer by four bytes, the first argument's address will be in EBP+8, the second argument's in EBP+12, the third's in EBP+16, and these offsets from EBP will not be altered by any local activity involving the ESP register.

For CHARACTER-valued arguments, the length of the string must also be passed. This is done by treating the lengths of the CHARACTER arguments as though they were extra parameters following the normal visible ones, and passing them by value. The term ‘by value’ in this context means that the actual length is pushed, rather then the address of the length. These length parameters are treated as though they were to the right of the actual parameters in the call, and so they are actually pushed first, and are at higher offsets relative to EBP.

Passing Arguments to Subroutines

It is often easiest to learn a programming method by studying examples, and so we will now show and examine several cases in which a Fortran program calls an assembly language subprogram.

First, the following main program (ADDMAIN) passes two INTEGER variables to a Fortran subroutine (FORADD), where they are added, with their sum being returned as a third variable.

Example 1: Simple Addition.

```
PROGRAM ADDMAIN
integer :: i,j,k,l
i = 17
j = 24
call foradd(i,j,k)
print *, 'i,j,k=',i,j,k
```

```
i = 52
j = 16
call asmadd(i,j,l)
print *, 'i,j,l=',i,j,l
stop
end
SUBROUTINE FORADD(II,JJ,KK)
kk = ii+jj
return
end
```

You should note that ADDMAIN also calls a second subroutine, ASMADD. Here it is:

```
             TITLE   ASMADD
             .386
             .MODEL  FLAT
;
_ACODE       SEGMENTPARA USE32 PUBLIC 'CODE'
             ASSUME  CS:_ACODE
             PUBLIC  _asmadd_      ; Entry point name
_asmadd_     PROC    NEAR          ; Start of procedure
             push    ebp           ; Save EBP
             mov     ebp,esp       ; Will use EBP for args
             push    ebx           ; Must save EBX
             mov     eax,[ebp+8]   ; 1st arg addr
             mov     ecx,[ebp+12] ; 2nd arg addr
             mov     edx,[ebp+16] ; 3rd arg addr
             mov     ebx,[eax]     ; 1st arg value
             mov     eax,[ecx]     ; 2nd arg value
             add     eax,ebx       ; Form I+J
             mov     [edx],eax     ; Store into K
             pop     ebx           ; Restore saved EBX
             mov     esp,ebp       ; Restore stack pointer
             pop     ebp           ; Restore base pointer
             ret                   ; Return to caller
_asmadd_     ENDP                  ; End of procedure
_ACODE       ENDS                  ; End of segment
             END
```

ASMADD is the assembly-language translation of FORADD: it also takes three variables, adds the first two, and returns the result in the third one. Examining ASMADD, we can see that once the preamble is completed, the addresses of the arguments are accessible to the assembly-language routine in EBP+8, EBP+12, and EBP+16. Since the EBX register is used in the processing, its contents must be preserved by being pushed onto the stack before it is clobbered, and popped off later.

LF95 assumes that the caller will fix the stack, i.e., remove the argument address pointers. As a result, the return to the calling routine is accomplished by means of a simple RET instruction.

Example 2: Using local data.

Now, let us examine a case in which a subroutine contains some local data. The main program MULMAIN calls two subroutines, FORMUL (written in Fortran), and ASMMUL written in assembly language. Both FORMUL and ASMMUL do the same thing: multiply the first argument by 7, add 3, and then return the result as the second argument. This is the Fortran part:

```
PROGRAM MULMAIN
integer :: i,j,k,l
i = 5
call formul(i,j)
print *, 'i,j=',i,j
k = 3
call asmmul(k,l)
print *, 'k,l=',k,l
stop
end
SUBROUTINE FORMUL(II,JJ)
jj = 7*ii + 3
return
end
```

Here is the assembly-language subroutine ASMMUL, with two constants m1 and m2 stored in a local data area.

```
             TITLE   ASMMUL
             .386
             .MODEL  FLAT
;
_ACODE       SEGMENT PARA USE32 PUBLIC 'CODE'
             ASSUME  CS:_ACODE, DS:_ADATA
             PUBLIC  _asmmul_     ; Entry point name
_asmmul_     PROC    NEAR         ; Start of procedure
             push    ebp          ; Save base pointer
             mov     ebp,esp      ; Save stack pointer
             mov     eax,[ebp+8]  ; 1st arg addr
             mov     eax,[eax]    ; 1st arg EAX=I
             mov     ecx, m1      ; 7 into ECX
             mul     ecx          ; 7*I is in EAX
             add     eax, m2      ; 7*I+3 is in EAX
             mov     edx,[ebp+12] ; 2nd arg addr
             mov     [edx],eax    ; Store in 2nd arg (J)
             mov     esp,ebp      ; Restore stack pointer
             pop     ebp          ; Restore base pointer
             ret
_asmmul_     ENDP
_ACODE       ENDS
;
_ADATA       SEGMENT PARA USE32 PUBLIC 'DATA'
```

```
m1            dd      7
m2            dd      3
_ADATA        ENDS
;
              END
```

The two variables are initialized to values of 7 and 3, and are not altered. Quantities stored in this manner could be changed during the course of computation, if required. Alternatively, this routine could have been written with the constants 7 and 3 being coded as immediate data in the MOV and ADD instructions that use them.

Example 3: Using floating-point arithmetic.

Floating point arithmetic is also possible in an assembly language routine that is called from an LF95 program. Here is an example of a main program (FLTMAIN) that calls two functionally-identical subroutines, FORFLT and ASMFLT, which are written in Fortran and assembly language, respectively.

```
   PROGRAM FLTMAIN
   real :: x, y, z
   x = 3.0
   y = 8.5
   call forflt(x,y,z)
   print 20, x,y,z
20 format (' x,y,z=',3F10.4)
   x = 4.5
   y = 7.1
   call asmflt(x,y,z)
   print 20, x,y,z
   stop
   end
   SUBROUTINE FORFLT(XX,YY,ZZ)
   zz = 3.1*xx + yy + 7.6
   return
   end
```

This is the assembly language routine, and we can see that REAL variables are also passed as addresses, located in EBP+8, EBP+12, EBP+16, etc.

```
              TITLE   ASMFLT
              .386
              .MODEL  FLAT
;
_ACODE        SEGMENT PARA USE32 PUBLIC 'CODE'
              ASSUME  CS:_ACODE, DS:_ADATA
              PUBLIC  _asmflt_        ; Entry point name
_asmflt_      PROC    NEAR            ; Start of procedure
              push    ebp             ; Save base pointer
              mov     ebp,esp         ; Save stack pointer
              mov     eax,[ebp+8]     ; Addr X
              mov     ecx,[ebp+12]    ; Addr Y
```

```
            mov     edx,[ebp+16]     ; Addr Z
            fld     dword ptr d1     ; Load 3.1
            fmul    dword ptr [eax] ; 3.1*X
            fadd    dword ptr [ecx] ; 3.1*X+Y
            fadd    dword ptr d2     ; 3.1*X+Y+7.6
            fstp    dword ptr [edx] ; Store into Z
            mov     esp,ebp          ; Restore stack pointer
            pop     ebp              ; Restore base pointer
            ret
_asmflt_    ENDP
_ACODE      ENDS
;
_ADATA      SEGMENT PARA USE32 PUBLIC 'DATA'
d1          dd      3.1
d2          dd      7.6
_ADATA      ENDS
;
            END
```

In assembly language, it is necessary to access the values of the variables using the keywords DWORD PTR for REAL(KIND=4) and QWORD PTR for REAL(KIND=8) variables.

Example 4: Using COMMON blocks.

If it is necessary for an assembly language subroutine to access the contents of a COMMON block, then we must find the starting address of that block.

The starting address of a named COMMON is put in a global variable; the name of that variable is composed by converting the COMMON block's name to lower case letters, and then attaching an underscore before and after the name. Thus, the starting address of a COMMON block that is named ZOOM can be found in the global variable _zoom_ . The starting address of blank COMMON is placed in the global variable __BLNK__. (Note that there are two underscore symbols both before and after the word ``BLNK.")

In the following example, both blank COMMON and COMMON/RRR/ are passed to a Fortran subroutine (FORCOM) and its assembly language equivalent (ASMCOM), where some minor calculations are performed.

```
      PROGRAM CMNMAIN
      common i,j,k
      common /rrr/ x,y,z
      i = 4; j = 17; k = 0
      x = 1.6; y = 3.7; z = 0.0
      call forcom
      print 10, i,j,k
   10 format (' i,j,k=',3I6)
      print 20, x,y,z
   20 format (' x,y,z=',3F10.4)
      i = 4; j = 17; k = 0
      x = 1.6; y = 3.7; z = 0.0
      call asmcom
      print 10, i,j,k
      print 20, x,y,z
      stop
      end
      SUBROUTINE FORCOM
      common i,j,k
      common /rrr/ x,y,z
      k = 5*i + j
      z = x*y
      return
      end
```

This is ASMCOM, the assembly language subroutine that manipulates variables in the two COMMON blocks.

```
             TITLE   ASMCOM
             .386
             .MODEL  FLAT
;
BLNKCOM      STRUCT
i            dword   ?
j            dword   ?
k            dword   ?
BLNKCOM      ENDS
;
             EXTERN   __BLNK__:BLNKCOM
;
RRRCOM       STRUCT
x            real4   ?
y            real4   ?
z            real4   ?
RRRCOM       ENDS
;
             EXTRN   _rrr_:RRRCOM
;
_ACODE       SEGMENTPARA USE32 PUBLIC 'CODE'
             ASSUME  CS:_ACODE, DS:_ADATA
             PUBLIC  _asmcom_        ; Entry point name
_asmcom_     PROC    NEAR            ; Start of procedure
             push    ebp             ; Save EBP
             mov     ebp,esp         ; use EBP for args
             mov     eax, dword ptr __BLNK__.i ; Get I
             mov     ecx, m1         ; Load 5
             mul     ecx             ; Form 5*I
             add     eax, dword ptr __BLNK__.j ; 5*I+J
             mov     dword ptr __BLNK__.k,eax ; Store into K
             fld     dword ptr _rrr_.x ; Load X
             fmul    dword ptr _rrr_.y ; Form X*Y
             fstp    dword ptr _rrr_.z ; Z=X*Y
             mov     esp,ebp         ; Restore stack pointer
             pop     ebp             ; Restore base pointer
             ret                     ; Return to caller
_asmcom_     ENDP                    ; End of procedure
_ACODE       ENDS                    ; End of segment
;
_ADATA       SEGMENT PARA USE32 PUBLIC 'DATA'
m1           dd      5
_ADATA       ENDS
;
             END
```

The starting addresses of the COMMON blocks are obtained by using EXTERN directives to connect to the global values. The individual variables within a COMMON block can then be accessed as STRUCTs that are written so as to match the layout of the Fortran code's COMMON declarations. Each COMMON block must consist of a STRUCT definition, plus an EXTERN declaration to connect it to the global data object.

Example 5: CHARACTER arguments.

Type CHARACTER variables are passed to subroutines as two arguments: the starting address of the string, and the string's length. The two arguments are not, however, pushed consecutively onto the stack. Rather, the address pointer is pushed in the usual order, and then after all arguments have been passed, the lengths of any CHARACTER arguments are passed by value.

Here is an example of a main program (CHRMAIN), that calls a Fortran subroutine (FORCAPS), and its assembly language equivalent (ASMCAPS). Both FORCAPS and ASMCAPS take two CHARACTER arguments; the first argument is converted into all upper case letters, and then returned in the second argument.

```
   PROGRAM CHRMAIN
   character (len=20) :: line1, line2, line3
   line1 = 'This is a message'
   line2 = 'zzzzzzzzzzzzzzzzzzzz'
   line3 = 'aaaaaaaaaaaaaaaaaaaa'
   call forcaps(line1,line2)
   print 20, line1
   print 20, line2
20 format (1X,A)
   call asmcaps(line1,line3)
   print 20, line1
   print 20, line3
   stop
   end
```

```
SUBROUTINE FORCAPS(L1,L2)
character*(*) :: l1, l2
n = len(l1)       ! Converts all
do i=1,n          ! chars to caps
  ic = ichar(l1(i:i))
  if (ic.ge.97 .and. ic.le.122) ic = ic-32
  l2(i:i) = char(ic)
enddo
return
end
```

This is the assembly language string capitalization routine.

```
             TITLE   ASMCAPS
             .386
             .MODEL  FLAT
;
_ACODE       SEGMENTPARA USE32 PUBLIC 'CODE'
             ASSUME  CS:_ACODE
             PUBLIC  _asmcaps_    ; Entry point name
_asmcaps_    PROC    NEAR         ; Start of procedure
             push    ebp          ; Save EBP
             mov     ebp,esp      ; Will use EBP for args
             push    esi          ; Must preserve ESI
             push    edi          ; Must preserve EDI
;
             mov     esi,[ebp+8]  ; 1st string addr (L1)
             mov     edi,[ebp+12] ; 2nd string addr (L2)
             mov     ecx,[ebp+16] ; 1st string length
             cmp     ecx, 0       ; Length nonpositive?
             jle     Exit         ; Yes, so return
;
Looper:      mov     al, [esi]    ; Get char from L1
             cmp     al, 97       ; Below "a"?
             jl      PutIt        ; Yes, so no conversion
             cmp     al, 122      ; Above "z"?
             jg      PutIt        ; Yes, so no conversion
             sub     al, 32       ; Change LC to UC
PutIt:       mov     [edi], al    ; Store
             inc     esi          ; Point to next char
             inc     edi          ; Point to next target
             loop    Looper       ; Loop until done
;
```

```
Exit:        pop     edi          ; Restore saved EDI
             pop     esi          ; Restore saved ESI
             mov     esp,ebp      ; Restore stack pointer
             pop     ebp          ; Restore base pointer
             ret                  ; Return to caller
_asmcaps_    ENDP                 ; End of procedure
_ACODE       ENDS                 ; End of segment
             END
```

Note that the starting addresses of the arguments are stored in EBP+8 and EBP+12, while the lengths of the two CHARACTER variables are in EBP+16 and EBP+20. In this code, we do not make use of the length of the second string, assuming it to be equal to that of the first one.

Since we use the ESI and EDI registers in this subroutine, we save their previous values on the stack and restore them before returning.

Returning Values from Functions

LF95 Function Conventions

The methods for passing arguments and COMMON blocks to a FUNCTION are identical to those described above for a SUBROUTINE. The only difference in the calling sequence is that a FUNCTION returns a value, and the method that is used to send the result back to the calling routine depends upon the data type of that value.

INTEGER-valued FUNCTIONs return values using CPU registers, so that the return value for one-byte, two-byte, and four-byte functions are returned in AL, AX, and EAX, respectively.

Four-byte and eight-byte REAL FUNCTIONs use the top of the floating-point unit stack, ST(0) for return of values. The only difference in the assembly language access of these variable types is that the former require DWORD PTR, while the latter use QWORD PTR when loading to and storing from the FPU. These conventions are summarized in Table 8 on page 91.

Table 8: FUNCTION Return Mechanisms

Function Type	Kind No.	Location of Return Value
INTEGER	1	AL
INTEGER	2	AX
INTEGER	4	EAX
LOGICAL	1	AL
LOGICAL	4	EAX
REAL	4	ST(0)
REAL	8	ST(0)
COMPLEX	4	Address on stack
COMPLEX	8	Address on stack
CHARACTER	all	Address & length on stack

Example 6: A COMPLEX Function

When an LF95 program calls a COMPLEX-valued function, it first pushes the argument addresses onto the stack, and then also pushes the address of a place where the function should store its return value. Thus, after the function preamble (where the contents of ESP are stored into EBP), EBP+8 will contain the address of the return buffer, and the normal argument pointers will start at EBP+12.

Here is an example of a program that passes a COMPLEX variable to a COMPLEX-valued Fortran function CXFFUN that returns two times its argument.

```
PROGRAM CXMAIN
complex :: a, b, c, cxffun, cxafun
a = (1.0,2.0)
b = cxffun(a)
c = cxafun(a)
print *, 'a=',a
print *, 'b=',b
print *, 'c=',c
stop
end
```

```
FUNCTION CXFFUN(A)
complex :: a, cxffun
cxffun = a+a
return
end
```

The above program also calls a COMPLEX-valued assembly language function CXAFUN, that performs exactly the same operation as CXFFUN, i.e., it returns double the argument.

```
            TITLE   CXAFUN
            .386
            .MODEL  FLAT
;
_ACODE      SEGMENT PARA USE32 PUBLIC 'CODE'
            ASSUME  CS:_ACODE
            PUBLIC  _cxafun_        ; Entry point name
_cxafun_    PROC    NEAR            ; Start of procedure
            push    ebp             ; Save EBP
            mov     ebp,esp         ; Will use EBP for args
;
            mov     eax, [ebp+12]   ; Argument address
            fld     dword ptr [eax] ; Get real part
            fadd    dword ptr [eax] ; Double it
            fld     dword ptr [eax+4] ; Get imag part
            fadd    dword ptr [eax+4] ; Double it
;
            mov     eax, [ebp+8]    ; Return buffer address
            fstp    dword ptr [eax+4] ; Store imag part
            fstp    dword ptr [eax] ; Store real part
;
            mov     esp,ebp         ; Restore stack pointer
            pop     ebp             ; Restore base pointer
            ret                     ; Return to caller
_cxafun_    ENDP                    ; End of procedure
_ACODE      ENDS                    ; End of segment
            END
```

Looking at this function, we can see that the single argument's address is stored in EBP+12. That is the address of the real part of the argument, with the imaginary part being stored four bytes higher in memory.

Both parts of the argument are copied into the FPU and doubled. The results are then stored into the return buffer, whose address is found at EBP+8. That is, of course, the address of the real part and the imaginary component is stored four bytes higher.

Example 7: A CHARACTER Function

A somewhat more complicated mechanism is used for CHARACTER-valued functions. After the argument information has been pushed on the stack, they are followed by the length and starting address of the memory buffer that will accept the result. As a consequence, the return buffer's address can be found in EBP+8, and its length in EBP+12. The address of the first argument is then moved up to EBP+16, and any other arguments follow in the usual manner.

Here is a Fortran main program that sets a CHARACTER variable equal to the string ``Hello," and then calls a Fortran function (FFUN) that returns a capitalized form of the string. The program then calls an assembly language function (AFUN) that returns a decapitalized version.

```
         PROGRAM CHMAIN
         character*20 a, b, c, ffun, afun
         a = 'Hello'
         b = ffun(a)
         c = afun(b)
         print 20, a, b, c
      20 format (' a = ',A/' b = ',A/' c = ',A)
         stop
         end
         CHARACTER*20 FUNCTION FFUN(A)
         character*(*) a
         n = len(a)
         do i=1,n
           ic = ichar(a(i:i))
           if (ic.ge.97 .and. ic.le.122) ic = ic-32
           ffun(i:i) = char(ic)
         enddo
         return
         end
```

This is the CHARACTER-valued assembly language function that is used by the program above:

```
             TITLE   AFUN
             .386
             .MODEL  FLAT
;
_ACODE       SEGMENT PARA USE32 PUBLIC 'CODE'
             ASSUME  CS:_ACODE
             PUBLIC  _afun_       ; Entry point name
_afun_       PROC    NEAR         ; Start of procedure
             push    ebp          ; Save EBP
             mov     ebp,esp      ; Will use EBP for args
             push    esi
             push    edi
;
             mov     edx, [ebp+12]; Length of return buffer
             mov     eax, [ebp+20]; Length of argument
             cmp     edx, eax     ; Which is smaller?
             jg      L10          ; Return buffer
```

```
                mov     ecx, edx      ; Get arg length
                jmp     L20
L10:            mov     ecx, eax      ; Get ret buf length
L20:            cmp     ecx, 0        ; Length nonpositive?
                jle     L90           ; Yes, so return
;
                mov     esi, [ebp+16]; Addr of argument
                mov     edi, [ebp+8] ; Addr of ret buf

L30:            mov     al, [esi]     ; Get char from L1
                cmp     al, 65        ; Below "A"?
                jl      L40           ; Yes, so no conversion
                cmp     al, 90        ; Above "Z"?
                jg      L40           ; Yes, so no conversion
                add     al, 32        ; Change UC to LC
L40:            mov     [edi], al     ; Store
                inc     esi           ; Point to next char
                inc     edi           ; Point to next target
                loop    L30           ; Loop until done
;
L90:            pop     edi           ; Restore saved EDI
                pop     esi
                mov     esp,ebp       ; Restore stack pointer
                pop     ebp           ; Restore base pointer
                ret                   ; Return to caller
_afun_          ENDP                  ; End of procedure
_ACODE          ENDS                  ; End of segment
                END
```

The sole argument is passed with its starting address in EBP+16, and its length in EBP+20 --- remember that if there are several arguments, then the CHARACTER lengths follow the entire list of addresses. The return buffer, the place where the function should store its return value is communicated by its starting address (in EBP+8) and length (in EBP+12).

4 Command-Line Debugging with FDB

FDB is a command-line symbolic source-level debugger for Fortran 95 and assembly programs. Before debugging your program you must compile it using the `-g` option (see*"Compiler and Linker Options"* on page 29). The `-g` option 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.

Run

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.

breakon [#n]

Enables the breakpoint number n. If #n is omitted, all breakpoints are enabled. Note that the "#" symbol is required.

breakoff [#n]

Disables, but does not remove, the breakpoint number n. If #n is omitted, all breakpoints are disabled. Note that the "#" symbol is required.

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.

stepi [*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.

silentstepi [*count*]

Same as Stepi 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* [= *value*]]

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, derived type element, or common block. *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

If *value* is specified, the variable will be set to *value*.

If no arguments are specified, the last print command having arguments is repeated.

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[tion] *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 sig encountered
`throw`	Execution not stopped when signal sig 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 105).

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 Library Manager

The Microsoft librarian utility, LIB, can be used to manage library creation and modification, extract object files from an existing library, or create import libraries. These three tasks are mutually exclusive, which means that LIB can only be invoked to perform one of these functions at a time.

By default, LIB outputs a file using the name of the first object or library file that is encountered, giving it the `.lib` extension. If a file with this name already exists, it is overwritten. The default action can be overridden by using the `/out:`*libname* option.

LIB accepts both OMF and COFF format object files. When an OMF object file is specified, LIB changes the format to COFF before creating a library.

LIB Syntax:

LIB *[options] [files]*

options is a list of options separated by spaces. Options begin with a hyphen (-) or a slash(/). They may appear in any order and are processed in the order they are encountered. Arguments to options are denoted by a colon character (:), and there cannot be any spaces or tabs between an option and it's argument.

files is a space separated list of object and library filenames.

Options

/CONVERT

Converts an import library to Visual Studio version 5 format.

/DEF[:*filename*]

Indicates that an import library is to be created. *filename* indicates the name of a definition file. LIB will export procedures that are specified in the EXPORT section a definition file or that are specified using the /EXPORT option.

/EXPORT:*symbol*

Used to specify procedures to be exported when creating an import library.

/EXTRACT:*membername*

Used to extract the object file *membername* from a library.

/INCLUDE:*symbol*

Adds *symbol* to the symbol table when creating an import library.

/LIBPATH:*dir*

Sets a path to be searched for library files. This path overrides a path specified by the LIB environment variable.

/LINK50COMPAT

Generates an import library using Visual Studio version 5 format.

/LIST[:*filename*]

Displays a list of objects in the first library file encountered. If *filename* is absent, the output is displayed on stdout. If *filename* is present, the output is directed to the specified file.

/NODEFAULTLIB[:*library*]

Do not refer to default libraries when resolving external references. If *library* is present, only the specified library is removed from the default library list.

/NOLOGO

Suppresses the display of the version and copyright banner.

/OUT:*libname*

Sets the name of the output library file.

/REMOVE:*membername*

Removes the object file named *membername* from the specified library.

/VERBOSE

Displays detailed information about the progress of the LIB session.

Response Files

It is possible to place commonly used or long LIB command-line parameters in a response file. LIB 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 separator on the LIB command line.

To invoke the response file, type:

LIB *@response-filename*

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

Creating and maintaining COFF libraries

The default usage for LIB is to perform library management. LIB runs in default mode whenever the `/def` or `/extract` options are not used. LIB will accept any object files and libraries specified on the command line and in a command file, and create a library containing the combined contents of the input files.

Example 1:

```
lib obj1.obj obj2.obj lib1.lib
```

In this example, the files `obj1.obj obj2.obj` and `lib1.lib` are combined into a library called `obj1.lib.` If `obj1.lib` did not exist before this command was invoked, it is created. If `obj1.lib` did exist before this command was invoked, it's previous contents are overwritten.

Example 2:

```
lib obj1.obj obj2.obj lib1.lib /out:mylib.lib
```

In this example, the files `obj1.obj obj2.obj` and `lib1.lib` are combined into a library called `mylib.lib.` If `mylib.lib` did not exist before this command was invoked, it is created. If `mylib.lib` did exist before this command was invoked, it's previous contents are overwritten.

Example 3:

```
lib /remove:obj1.obj mylib.lib
```

In this example, the object file `obj1.obj` is removed from the library `mylib.lib.`

Example 4:

```
lib mylib.lib obj1.obj
```

In this example, the object file `obj1.obj` is added to the library `mylib.lib.`

Extracting object files from libraries

When the /extract option is used, LIB extracts an object file from an existing library. The object being extracted is not removed from the library. To delete an object from a library use the /remove option.

Example:

```
lib /extract:obj1.obj mylib.lib
```

In this example, the object file obj1.obj is extracted from the library mylib.lib and is written to disk. If a file named obj1.obj previously existed, it is overwritten.

Creating import libraries

When the /def option is specified, LIB is used to generate an import library. It is usually not necessary to use LIB to create an import library, because the import library is automatically generated by LINK whenever a DLL is created. If the user is creating a DLL with a 3rd party language system and an import library is not created, or if the user is provided with a DLL by a 3rd party without an import library, one can be generated using LIB /def. For more information on creating import libraries for mixed language applications, see *"Building Import Libraries"* on page 57.

Two items are needed to generate an import library - a set of definitions and an object file containing the exported procedures.

Definitions may be in the form of a definition file or as arguments to the /EXPORT option. A definition file contains exported symbols as they appear in the DLL. These symbols can be listed using DUMPBIN /EXPORTS. Alternatively, a definition file can be generated from a DLL using the MAKEDEF utility. Note that the definition file that is used when creating the import library should only contain procedure names that appear in the object file, otherwise unresolved references will occur when the LIB command is executed.

If the object file that was used to create the DLL is available, an import library can easily be created using the object file and a definition file.

Example:

```
lib /def:mydll.def dllobj.obj /out:mydll.lib
```

In this example the file mydll.def contains an EXPORTS header, under which export symbols are listed as they appear when displayed with the DUMPBIN utility. The file dllobj.obj is the object file that was linked to make the DLL.

If no object file is available, a Fortran object file can be created from a Fortran 'stub'. All that is required is that the user know the calling sequence for the DLL procedure. A stub procedure consists of a SUBROUTINE or FUNCTION statement, an argument list, declarations for any dummy arguments, a DLL_EXPORT statement, and an END statement. Note that the stub procedure name appearing in the DLL_EXPORT statement is case-sensitive, and should have the same case as the procedure exported from the DLL. The stub procedure is compiled into an object file using the `-c` and an appropriate `-ml` option. This object file can then be used by `LIB` to create the import library. When compiling the LF95 program that will call the DLL, make sure that the same `-ml` option is used as for the stub procedure. Note that `-ml lf95` is not a valid option when making an import library.

Example stub procedure (called dllsub1.f90):

```
subroutine dllsub1(a,i,l)
  dllexport :: dllsub1
  real      :: a
  integer   :: i
  logical   :: l
end subroutine
```

Example definition file (called dllsub1.def):

```
EXPORTS
dllsub1
```

Example compile command:

```
lf95 -c -ml msvc dllsub1.f90
```

Example LIB command:

```
lib /def:dllsub1.def dllsub1.obj /out:mydll.lib
```

The above examples show how to create an import library from a Fortran stub for a DLL called `mydll.dll,` which contains a procedure called `dllsub1` having three arguments. When compiling the LF95 main program which calls `mydll.dll`, the `-ml msvc` option must be used.

Note that depending on which target is specified when using the `-ml` option, `LIB` may generate a warning about multiply defined symbols. This warning can generally be disregarded.

Further examples of creating import libraries using `/def` and stub procedures exist in directories under the `EXAMPLES\MIX_LANG` directory.

6 Utility Programs

This chapter documents the following utility programs:

- DUMPBIN.EXE
- EDITBIN.EXE
- HDRSTRIP.F90
- LFSPLIT.EXE
- MAKEDEF.EXE
- SEQUNF.F90
- TRYBLOCK.F90
- UNFSEQ.F90
- WHERE.EXE

DUMPBIN.EXE

DUMPBIN.EXE allows you to display information about COFF object files, libraries of COFF object files, executable files, and dynamic-link libraries. Information can be displayed in both hexadecimal and ASCII character formats.

Invoking DUMPBIN

DUMPBIN is invoked from the command prompt using the following syntax:

dumpbin *[options] files*

DUMPBIN Options

Options are distinguished by using an option specifier, which consists of a leading “/” or “-” character, followed by the option name. Options and filenames may be separated by the space or tab characters. Options and filenames are not case sensitive. If no options are specified, the default option is /SUMMARY.

Option list

Note that some options for DUMPBIN may not apply to files built with LF95. Only options known to be valid for files built with LF95 are described.

-ALL

Displays everything except disassembly. Use /RAWDATA:NONE with the /ALL option to prevent display of raw binary details.

-ARCHIVEMEMBERS

Displays information about objects in a library.

-DEPENDENTS

Displays the name of any DLL needed by an executable or DLL.

-DISASM

Displays code disassembly.

-EXPORTS

Displays all symbols exported by a DLL.

-HEADERS

Displays coff header information.

-IMPORTS

Displays all symbols imported by an executable or DLL.

-LINKERMEMBER*[:lev]*

Displays public symbols defined in a library. If the *lev* argument is 1, display symbols in object order, along with their offsets. If the *lev* argument is 2, display offsets and index numbers of objects, then list the symbols in alphabetical order along with the object index for each. If the *lev* argument is not present, both outputs are displayed.

-OUT:*filename*

Sends output to the specified file instead of to the console.

-RAWDATA:*option*

Displays the raw contents of each section in the file. The *option* argument controls the format of the display, as follows:

BYTES - Default setting. Contents are displayed in hexadecimal bytes, and in ASCII.
SHORTS - Contents are displayed in hexadecimal words.
LONGS - Contents are displayed in hexadecimal long words.
NONE - Display of raw data is suppressed.
number - Controls the number of values displayed per line.

-RELOCATIONS

Displays any relocations in the object or image.

-SECTION:*section*

Restricts output to the specified section.

-SUMMARY
Default option. Displays minimal information about the file.

-SYMBOLS
Displays the COFF symbol table for an object file or library.

EDITBIN.EXE

EDITBIN.EXE allows you to edit information in COFF object files, libraries of COFF object files, executable files, and dynamic-link libraries. EDITBIN can also be used to convert object model format files (OMF) to common object file format (COFF). To convert from OMF to COFF, run EDITBIN with no options.

Invoking EDITBIN

EDITBIN is invoked from the command prompt using the following syntax:

editbin *[options] files*

EDITBIN Options

Options are distinguished by using an option specifier, which consists of a leading “/” or “-” character, followed by the option name. Options and filenames may be separated by the space or tab characters. Options and filenames are not case sensitive.

Option list

-BIND*[*:PATH=*path]*
Sets the addresses of the entry points in the import address table for an executable file or DLL. Use this option to reduce load time of a program. The optional *path* argument specifies the location of any DLLs. Separate multiple directories with semicolons. If *path* is not specified, EDITBIN searches the directories specified in the PATH environment variable. If *path* is specified, EDITBIN ignores the PATH variable.

-HEAP:*reserve[,commit]*
Sets the size of the heap in bytes. Numbers are specified in decimal format.

The *reserve* argument specifies the total heap allocation in virtual memory. The default heap size is 1MB. The linker rounds the specified value up to the nearest 4 bytes.

The optional *commit* argument specifies the amount of physical memory to allocate at a time. Committed virtual memory causes space to be reserved in the paging file. A larger *commit* value saves time when the application needs more heap space but increases the memory requirements and possibly startup time.

-LARGEADDRESSAWARE
Edits the image to indicate that the application can handle addresses larger than 2 gigabytes.

-NOLOGO
Suppresses display of the EDITBIN copyright message and version number.

-REBASE*[:modifiers]*
Sets the base addresses for the specified files. Assigns new base addresses in a contiguous address space according to the size of each file rounded up to the nearest 64K. Numbers are specified in decimal format. One or more optional *modifiers* are separated by a comma:

BASE=*address* - Beginning address for reassigning base addresses to the files. If BASE is not specified, the default starting base address is 0x400000. If DOWN is used, BASE must be specified, and *address* sets the end of the range of base addresses.

BASEFILE - Creates a file named COFFBASE.TXT, which is a text file in the format expected by LINK's /BASE option.

DOWN - Reassign base addresses downward from an ending address. Files are reassigned in the order specified, with the first file located in the highest possible address below the end of the address range. BASE must be used with DOWN to ensure sufficient address space for basing the files. To determine the address space needed by the specified files, run EDITBIN with the /REBASE option on the files and add 64K to the displayed total size.

-RELEASE
Sets the checksum in the header of an executable file.

-SECTION:*name[=newname][,properties][,alignment]*
Changes the properties of a section, overriding the properties that were set when the object file for the section was compiled or linked. *properties* and *alignment* characters are specified as a string with no white space.

name is the name of the section to modify.

newname is the new section name.

properties is a comma separated list of characters. To negate a property, precede its character with an exclamation point (!). The following properties may be specified:

c	- code
d	- discardable
e	- executable
i	- initialized data
k	- cached virtual memory
m	- link remove
o	- link info
p	- paged virtual memory
r	- read
s	- shared
u	- uninitialized data
w	- write

alignment is specified by the character "a" followed by a character to set the size of alignment in bytes, as follows:

1	- 1 byte
2	- 2 bytes
4	- 4 bytes
8	- 8 bytes
p	- 16 bytes
t	- 32 bytes
s	- 64 bytes
x	- no alignment

-STACK:*reserve[,commit]*

Sets the size of the stack in bytes. Numbers are specified in decimal format. The /STACK option applies only to an executable file.

The *reserve* argument specifies the total heap allocation in virtual memory. The default heap size is 1MB. The linker rounds the specified value up to the nearest 4 bytes.

The optional *commit* argument specifies the amount of physical memory to allocate at a time. Committed virtual memory causes space to be reserved in the paging file. A larger *commit* value saves time when the application needs more heap space but increases the memory requirements and possibly startup time.

-SUBSYSTEM:*system[,major[.minor]]*

Edits the image to indicate which subsystem the operating system must invoke for execution.

Tells the operating system how to run the executable file. *system* is specified as follows:

CONSOLE	- Used for Win32 character-mode applications.
WINDOWS	- Used for applications that do not require a console.

The optional *major* and *minor* version numbers specify the minimum required version of the subsystem.

-VERSION:*left[,right]*

Places a version number into the header of the image.
left indicates the portion of the version number that appears to the left of the decimal point.
right indicates the portion of the version number that appears to the right of the decimal point.

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.

LFSPLIT.EXE

Run LFSPLIT.EXE to divide a source file into new separate source files, one for each main program, subroutine, function or module. Each new source file will have a filename of the sub-program unit name and the same extension as the original file.

Type `lfsplit -help` at the command prompt for more details about use of the file splitter.

MAKEDEF.EXE

Use MAKEDEF.EXE to create a definition file listing all exported symbols from a DLL. The definition file is used by LIB.EXE to create an import library. MAKEDEF accepts a single DLL file including the .dll extension as a command line argument, and creates a file with the same name having the .def extension. If a definition file with this name already exists, it is overwritten. MAKEDEF ignores all exported symbols that contain three or more sequential underscore characters. The MAKEDEF utility requires that DUMPBIN.EXE be available in a directory on the path. See *"Creating import libraries"* on page 112 for instructions on generating an import library.

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 blocksizes and displays an elapsed time for I/O operations with each blocksize. 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.F90

UNFSEQ.F90 is a program that converts LF95 unformatted sequential files to LF90 style.

WHERE.EXE

WHERE.EXE can be used to locate files on the path or in directories, and to display the exe type, time and size of the file.

Invoking WHERE

`WHERE` is invoked with the following syntax:

WHERE *[*/r *dir] [*/Qqte*] pattern ...*

Where:

/r *dir* recursively search directories under *dir*
/Q display output files in double quotes
/q quiet mode, exit code of zero indicates file found
/t display file size and time
/e display executable type
pattern is one or more file specifications, with the wildcards, * ?, allowed

If /r is not specified, `WHERE` searches along the path.

Examples

```
where lf95.exe
```

Searches along the path for all occurrences of LF95.

```
where /te /r \windows user32.dll
```

Recursively searches all directories under `\windows` for all occurences of USER32.DLL, and lists each file size and creation time, and the executable type.

A 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:

- 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, with no header record. 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

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` option.

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 `-Wl,Ry,li` for example,

```
       myprog.exe -Wl,Ry,li
```

For more information on runtime options, see *"Runtime Options"* on page 129.

Limits of Operation.

Table 9: LF95 Limits of Operation

Item	Maximum
program size	4 Gigabytes or available memory (including virtual memory), whichever is smaller
number of files open concurrently	Not limited by LF95 language system.
Length of CHARACER datum	2,147,483,647 bytes
I/O block size	65,000 bytes
I/O record length	2,147,483,647 bytes
I/O file size (except transparent access)	18,446,744,073,709,551,614 bytes
I/O file size (transparent access)	4,294,967,296 bytes
I/O maximum number of records for direct access and transparent access files	2,147,483,647
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 9: LF95 Limits of Operation

Item	Maximum
number of array dimensions	7
array size	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: $T = l1 \times s + \sum_{i=2}^{n} \left\{ li \times \left(\prod_{m=2}^{i} dm - 1 \times s \right) \right\}$ *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

B 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 returned by the GETCL, GETPARM, and 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 10: 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 *[*,C*unit] [*,M*] [*,Q*] [*,R*e] [*,Rm:*file] [*,T*unit] [*,a*]* *[*,d*num] [*,e*num] [*,g*num] [*,i*] [*,l*elvl] [*,m*unit] [*,n*][*,p*unit] [*,q*] [*,r*unit] [*,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 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 and 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 /Wl,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 /Wl,Re
```

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 /Wl,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 /Wl,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 FU*nn*BF where *nn* is the unit number (see*“Shell Variables for Input/Output”* on page 136 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 e*num* 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
```

g*num*

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 fu*xx*bf, where *xx* is the unit number (see *"Shell Variables for Input/Output"* on page 136 for details). When the g option is specified at the same time as the shell variable fu*xx*bf, 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

Example:

```
a.exe /Wl,i
```

l*errlvl* *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
```

m*u_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 136 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
```

p*u_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 136 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 `-nfix` is in effect, the characters appear in uppercase letters so the q option is not required.

Example:

```
a.exe /Wl,q
```

r*u_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 136 for further details.

Example:

```
a.exe /Wl,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 /Wl,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 /Wl,x
```

Shell Variables for Input/Output

This section describes shell variables that control file input/output operations

FU*nn* = *filname*

The FU*nn* 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*nn*BF = *size*

The FU*nn*BF shell variable specifies the size of the input/output work area used by a sequential or direct access input/output statement. The value *nn* in the FU*nn*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

INDEX