# Lahey/Fujitsu Fortran 95 User's Guide Linux Edition

*Revision D*

**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**
**support@lahey.com**

# Table of Contents

# 1 Getting Started

Lahey/Fujitsu Fortran 95 (LF95) is a set of software tools for developing optimized 32-bit Fortran applications. LF95 is a complete implementation of the Fortran 95 standard. Lahey provides two distributions of LF95, called LF95 Linux Express and LF95 Linux PRO. Some chapters or feature descriptions in this manual apply only to LF95 Linux PRO. These chapters and feature descriptions are marked “PRO Version Only”.

## LF95 Linux Express

LF95 Express includes an optimizing compiler, debugger, on-line documentation and free e-mail technical support. Express has two manuals, the *User’s Guide* (this manual), which describes how to use the compiler and tools, and the *Language Reference*, which describes the Fortran 95 language and various extensions.

## LF95 Linux PRO

LF95 PRO includes an optimizing compiler with automatic parallelization and OpenMP support, debugger, AUTOMAKE (an automatic build tool for Fortran and C), WiSK (an X-Windows-based user interface and graphics toolset library), hardcopy manuals and free telephone support. PRO documentation includes the *User’s Guide*, the *Language Reference*, and the *WiSK Reference*, which documents the use of the Winteracter Starter Kit (WiSK) for graphics and user interface development.

This manual assumes that the reader possesses a working knowledge of the Linux operating system, including Linux commands, file manipulation, file system navigation, and shell scripts.

## System Requirements

- An 80486DX, Pentium series or compatible processor
- 24 MB of RAM (32 MB or more recommended)
- 60 MB of available hard disk space for LF95 Linux PRO; 30 MB for LF95 Linux Express
- X-Windows to use WiSK and view the online PDF documentation

- A compatible version of the Linux operating system. Table 1 shows the versions of Linux that are known to be compatible with LF95. Other Linux variants might be compatible if they include kernel version 2.4.7 or later and libc version 2.2.4 or later (see `README` for last minute updates):

**Table 1: Supported Distributions**

| Distribution | Kernel | libc |
|---|---|---|
| Red Hat Linux 7.2 | 2.4.7 | 2.2.4 |
| Red Hat Linux 7.3 | 2.4.18 | 2.2.5 |
| Red Hat Linux 8.0 | 2.4.18 | 2.2.93 |
| Slackware v8.1 | 2.4.18 | 2.2.5 |
| Linux Mandrake v9.0 | 2.4.19 | 2.2.5 |
| SuSE Linux 8.1 | 2.4.19 | 2.2.5 |

## Manual Organization

This book is organized into seven chapters and three appendices.

- Chapter 1, *Getting Started*, identifies system requirements, describes the installation process, and takes you through the steps of building your first program.
- Chapter 2, *Developing with LF95*, describes the development process and the driver program that controls compilation, linking, and the generation of executable programs or libraries.
- Chapter 3, *Mixed Language Programming*, describes the creation of mixed language programs using C or G77.
- Chapter 4, *Command-Line Debugging with fdb*, describes the command-line debugger.
- Chapter 5, *Multi-Processing (PRO version only)*, describes how to use LF95 PRO’s automatic and OpenMP parallelization capabilities.
- Chapter 6, *Automake (PRO version only)*, describes how to use Automake, LF95 PRO’s automatic build tool.

- Chapter 7, *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.
- Appendix C, *Lahey Technical Support* describes the services available from Lahey and what to do if you have trouble.

## Notational Conventions

The following conventions are used throughout this manual:

`Code` and `keystrokes` are indicated by courier font.

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

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

*Italics* indicate text to be replaced by the programmer.

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

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.

## Product Registration

To all registered LF95 Express users, Lahey provides free, unlimited technical support via fax, postal mail, and e-mail. Registered LF95 PRO users are additionally entitled to free phone support. Procedures for using Lahey Support Services are documented in Appendix C, *Lahey Technical Support*.

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

## Installing Lahey/Fujitsu Fortran 95

In order to install LF95 in a public directory, you must be logged in as root. The installation CD must be mounted with execute permission. The install script presents a series of choices, which guide the user through the installation process.

3. Run `install`, the installation script, and follow the menu prompts. The default installation directory is `/usr/local/lf9561`, however, you can change it to a directory of your choice during the installation.

4. If desired, you may install the Adobe Acrobat Reader at a later time. You may run `install` and select it from the menu or install it manually. It is located in the `acrobat` directory on the installation CD as a compressed tar file. Acrobat Reader or xpdf is required to view the on-line documentation.

## Maintenance Updates

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

To get the latest maintenance update for this version, go to Lahey’s web page:

**http://www.lahey.com**

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.

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.

## Uninstalling LF95

The uninstallation program can be found in the LF95 installation directory (/usr/local/lf95xx by default). You must be logged in as root in order to uninstall LF95. Any new files created after installation will not be removed.

## Building Your First LF95 Program

Building and running a Fortran program with LF95 involves three basic steps:

1. Creating a source file using a text editor.

2. Generating an executable program using the LF95 driver. The 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.

## 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 driver program. From the command prompt, build the demo program by changing to the directory where `demo.f90` is installed (located in `examples/fortran/` under the installation directory), and entering

```
lf95 demo.f90
```

This causes the compiler to read the source file `demo.f90` and compile it into the object file `demo.o`. Once `demo.o` is created, LF95 invokes the linker to combine necessary routines from the runtime library and produce the executable program, `a.out`.

## Running the Program

To run the program, type its name at the command prompt and press `<Enter>`:

```
./a.out
```

The `demo` program begins and a screen similar to the following is displayed:

```
                    Lahey/Fujitsu LF95 Compiler
                    ---------------------------

           installation test and demonstration program

                      Copyright(c) 1998
                 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` and `ERRATA`. These contain important last-minute information and changes to the documentation.

## Other Sources of Information

### Files

| | |
|---|---|
| `README` | last-minute information |
| `FILELIST` | description of all files distributed with LF95 |
| `RTERRMSG` | descriptions of runtime error messages and their `IOSTAT` values |
| `ERRATA` | changes that were made after the manuals were finalized |

### Manuals

*Lahey/Fujitsu Fortran 95 Language Reference*
*Winteracter Starter Kit Reference* (PRO Version only)

### Newsletters

The Lahey Fortran *Source* newsletter

### Lahey Web Page

`http://www.lahey.com`

### Discussion Groups

The Lahey Fortran Forum (see Lahey Web Page for instructions on joining this discussion group)

# 2 Developing with LF95

This chapter describes how to use LF95's driver to build Fortran applications. The driver controls compilation, linking, and the production of archive libraries, executable programs and shared libraries.

## How the Driver Works

The driver (`lf95`) controls the two main processes—compilation and linking—used to create an executable program. These component 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 95 modules. It is this component that performs the actual compilation of the program, even though the `lf95` driver is commonly referred to as the "compiler."

**Linux Archive Utility**. `ar`, the archive utility, can be invoked from the driver or from the command prompt to create or change static libraries.

**Linux Linker.** `ld` is the linker. The linker combines object files and libraries into a single executable program or shared library.

## Running the LF95 Driver

By default, the LF95 driver program oversees compilation of any specified source files and will 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 filenames and optional command-line options:

`lf95` *[options] filenames [options]*

The driver searches for the various tools (the compiler, archive library utility, and linker) first in the directory the driver is located and then, if not found, on your path. The command line options are discussed later in this chapter.

## Filenames and Extensions

Depending on the extension(s) of the filename(s) specified, the driver will invoke the necessary tools. The extensions `.f95`, `.f90`, `.for`, `.f`, `.F95`, `.F90`, `.FOR`, and `.F`, for example, cause the compiler to be invoked. The extension `.s` causes the assembler to be invoked. The extension `.o` (denoting an object file) causes the linker to be invoked. Please note that if the suffix for Fortran source is uppercase (`.F95`, `.F90`, `.FOR`, or `.F`), it will cause the C preprocessor to be invoked before the compiler; it is therefore preferable to use a lowercase extension on the filename if the file does not need to be preprocessed.

For lowercase suffixes, the C preprocessor can be invoked using the `-Cpp` option. Preprocessor options `-D` (define macro), `-U` (un-define macro), and `-P` (send preprocessor output to file) are also supported, and behave as documented in the man pages for `gcc`, the GNU C-compiler. This manual does not encourage use of the preprocessor, because such activity fosters non-Fortran-standard programming practices.

**Please note:** filenames are case sensitive. Filenames containing spaces are not recommended, nor are filenames beginning with a hyphen, i.e., "`-`". Also note that 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 usual Linux wild-card characters. Filenames must be separated by a space. Filenames not matching any of the forms described below are passed directly to the linker.

#### 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, `a.out,` would be created in the current directory. `three.for` would not be compiled because its extension does not match the extension specified on the LF95 command line.

A source filename must be specified completely, including the extension. In the absence of an option specifying otherwise (i.e., if neither `--fix` or `--nfix` is specified):

`.f90, .F90, .f95, and .F95` specify interpretation as Fortran 95 free source form.

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

Once again, please note that an uppercase extension will cause the C preprocessor to be invoked before the Fortran compiler is invoked; it is therefore preferable to use a lowercase extension on the filename, if the file does not need to be preprocessed. For a description of free source form and fixed source form, please see the Language Reference.

### Object Filenames

The default name for an object file is the same as the source filename with extension `.o`. When an object file is created, it is placed by default in the current working directory. This behavior may be overridden by specifying the -o (or --out) option with a new name and path (see *“-o name”* on page 24).

### Module Filenames

Files containing Fortran 95 module information will have the same name as the module defined in the source code, in lowercase, followed by the `.mod` extension. When a module file is created, it is placed by default in the current working directory. This behavior may be overridden by specifying the `--mod` or `-M` option (see *“-M dir”* on page 23). The extension `.mod` is reserved for compiler-generated module files. Do not use this extension for your Fortran source files. If a program contains code that USEs a module, then its object file (corresponding to the source file where that module was defined) must be specified on the command line. The search path for `.mod` files may be specified with the `--mod` or `-M` option.

### Output Filenames

The default name for the executable file produced by the driver is `a.out`. If no path is specified, the current directory will be used. This may be overridden by specifying the `--out` or `-o` option with a new name and path. When -c is specified on the command line, the argument to `--out` or `-o` must be an object filename. (see *“-o name”* on page 24).

### Library Filenames

The default name for a library typically has an extension of `.a` for a static library and `.so` for a shared (dynamic) library (See *“Archive Libraries”* and *“Shared Libraries”* on page 12). In addition, libraries will typically begin with the characters “`lib`.” The prefix and extension must be omitted when referencing the library at link time. For example, `libsub.so` is a shared library in the current directory that is referenced on the command line as

```
lf95 main.f90 -L. -lsub
```

## Options

The driver recognizes one or more letters preceded by one or two hyphens (`-` or `--`) 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. **Please note:** options with double hyphens (--) require a delimiting space between the option and its argument(s); however, options with single hyphens (-) may be followed immediately by the argument(s), with no intervening space. If an option has multiple arguments, spaces are not allowed between the arguments.

**Example**

`-M../MyDir/IncDir` *(delimiting space not required)*

`--mod ../MyDir/IncDir:./ModDir`

*(delimiting space required after --mod but prohibited after :)*

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. Even though options with double hyphens are not case-sensitive, it is recommended that all options be treated as case-sensitive to avoid confusion. Certain arguments to driver options (i.e., names of files or directories) will also be case-sensitive. To illustrate, if the argument to the `-M` option in the above example were changed to `../MYDIR/INCDIR`, then the driver would be unable to find the actual directory.

An option for another component tool (linker, assembler, or preprocessor) that conflicts with an LF95 option may be passed directly to that component, verbatim, using the `-Wl`, `-Wa`, and `-Wp` options. These options behave as documented in the man pages for `gcc`, the GNU C compiler.

### 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 --lst --nlst`, the `--nlst` option would be used.

To display the LF95 version number and a summary of valid command-line options, type `lf95 --version --help`.

## Driver Configuration File (lf95.fig)

In addition to specifying options on the command line, you may specify a default set of options in the `lf95.fig` file. When the driver is invoked, the options in the `lf95.fig` file are processed before those on the command line. Command-line options override those in the `lf95.fig` file. The driver searches for `lf95.fig` first in the current directory and then, if not found, in the directory in which the driver is located.

## Command Files

If you have too many 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 an `@` character. When lf95 encounters a filename that begins with `@` on the command line, it opens the file and processes the commands in it.

**Example**

```
lf95 @mycmds
```

In this example, lf95 reads its commands from the file `mycmds`.

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

## Intermediate Files

The lf95 driver (and the components it controls) may use temporary files for storing intermediate results and passing them between components. These files are automatically created in the default temporary directory, using random names, and then deleted. This directory can be changed by specifying a value for the shell variable `TMPDIR`.

## Return Codes from the Driver

When the lf95 driver receives a failure return code, it aborts the build process. The driver will return an error code depending on the success of the invoked tools. These return codes are listed below:

**Table 2: Driver Return Codes**

| Code | Condition |
|---|---|
| `0` | Successful compilation and link |
| `1` | Compiler fatal error |
| `2` | Archive Utility error |
| `3` | Linker error |
| `4` | Driver error |

## Shared Libraries

A shared library is a collection of procedures packaged together in a library that is loaded at runtime. On Unix systems, such libraries have been traditionally referred to as “shared libraries” or “shared archives”. The term “DLL” (Dynamic Link Library) was coined as a name for the Microsoft Windows implementation of shared libraries. This manual uses the term “shared library” rather than “DLL,” even though the two can be considered as interchangeable. A shared library cannot run on its own; the functions and subroutines in a shared library must be called from an executable file that contains a main program. If an LF95 program that uses shared libraries is distributed to other machines, the shared libraries it uses must also be distributed or made available at runtime (see *“Distributing LF95 Applications”* on page 33).

## Archive Libraries

An archive library (sometimes called a “static library,” or simply an “archive”) is a collection of procedures in object form, stored in a file that may be referenced by the linker. At link time, when the executable program is created, the object code for procedures needed from the library by the program is incorporated into the program’s executable file.

## Using Shared Libraries

To create a shared library, use the `--shared` option.

**Example**

```
lf95 sub.f90 --out libsub.so --shared

lf95 main.f90 -L. -lsub
```

In this example, the source file `sub.f90` contains subroutines or functions, and the source file `main.f90` contains references to these procedures. The following takes place:

1. `sub.f90` is compiled to create object file `sub.o`.
2. `sub.o` is linked to create `libsub.so,` a shared library. Object file `sub.o` is then deleted.
3. `main.f90` is compiled to create `main.o`.
4. `main.o` is linked with the LF95 runtime library and combined with dynamic link information, referencing procedures in `libsub.so,` to create an executable program. Object file `main.o` is then deleted.

Note that the name of the shared library must be prefixed with “`lib`.” Also note that at runtime, `libsub.so` must be available on one of the directories specified in the `LD_LIBRARY_PATH` variable.

## Using Archive Libraries

To create an archive library, use the `--nshared` option.

**Example**

```
lf95 sub.f90 --out libsub.a --nshared

lf95 main.f90 -L. -lsub
```

Using the same source files as in the example above, The following takes place:

1. `sub.f90` is compiled to create `sub.o`.
2. the archive utility, ar, is automatically invoked to create `libsub.a` from `sub.o`. Note that `libsub.a` is an archive (static) library.
3. `main.f90` is compiled to create `main.o`.
4. `main.o` is statically linked with the necessary object code contained in `libsub.a` to create an executable program. Note that shared library `libsub.so` must not be present in the current directory; otherwise the linker will try to reference that file instead (See “Linking Rules” on page 32.).

## 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 objects and/or modules are created (if necessary). See *"-[n]c"* on page 16. Otherwise, processing continues with linking and creation of the executable program or library file.

### 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 --nwarn 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 21).

If no unrecoverable or serious errors are detected by the compiler, the error return code is set to zero (see *"Return Codes from the Driver"* on page 12). Unrecoverable or serious errors detected by the compiler (improper syntax, for example) terminate the build process. An object file is not created.

## Compiler and Linker Options

You can control compilation and linking by using any of the following option options. Options that use a single hyphen are case-sensitive. Some options apply only to the compilation phase, others to the linking phase, and still others (such as `-g`) 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.

Most LF95 options begin with two hyphens and are self-descriptive. Commonly used single-hyphen options are provided (`-I`, `-l`, `-L`, `-g`, `-o, -O, -c`, etc.) for compatibility with other Linux products (see descriptions below).

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 skipped, and all linker options will be ignored.

While linking is ultimately performed by `ld`, the GNU linker, it is best to perform linking of LF95 objects using the LF95 driver. This will help to insure that all necessary steps are taken and all necessary components are included to produce the final product. Any options not recognized by the LF95 driver will be passed directly to `ld`. Remember that any options passed directly to `ld` will be treated as case sensitive.

### --*[*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 I/O block size**

Compile only. Default: 8192 bytes

Specify --block to change the default block size on OPEN statements. See “*BLOCKSIZE=*” in the LF95 Language Reference. *blocksize* must be a decimal INTEGER constant. Specifying an optimal *blocksize* can make a significant improvement in the speed of your executable. The program `tryblock.f90` demonstrates how changing blocksize can affect execution speed. Some experimentation with *blocksize* in your program is usually necessary to determine the optimal value. This optimal value varies from one machine to the next; therefore, if your program is moved to another machine and optimal performance is desired, then *blocksize* should be re-evaluated.

## -*[*n*]*c

**Suppress Linking**

Compile only. Default: `-nc` (or `-c` not present)

Specify -c to create object (`.o`), and, if necessary, module (`.mod`) files without creating an executable. This is especially useful in makefiles, where it is not always desirable to perform the entire build process with one invocation of the driver.

## --*[*n*]*chk

**Checking**

Compile only. Default: `--nchk`

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

**Syntax**

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

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

**Table 3: --chk Arguments**

| Diagnostic Checking Class | 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.

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 --O0.  --chk overrides --parallel.

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

- The referenced expression has the POINTER attribute or is a structure one or more of whose structure components has the POINTER attribute.
- The referenced expression is an assumed-shape array.
- The referenced expression is an array section with vector subscript.
- The referenced variable is a dummy argument corresponding to an actual argument that is an array section.
- The referenced expression is in a masked array assignment.
- The derived type variable with an ultimate component that is an allocatable array.
- 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 has the SAVE attribute.
- The derived type variable with an ultimate component that is an allocatable array.

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: -9.0818487627532284154072898964213742q-4043
Default complex: (-5.37508134e-32,-5.37508134e-32)

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

Quadruple precision complex: (-9.0818487627532284154072898964213742q-4043, -90818487627532284154072898964213742q-4043)

Character : Z’8B’

**Example**

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

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

## --*[*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.

The --chkglobal option will not check bounds in the following conditions:

- The referenced expression has the POINTER attribute or is a structure one or more of whose ultimate 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.

### --*[*n*]*co

**Compiler Options**

Compile and link. Default: `--nco`

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

### --*[*n*]*dal

**Deallocate Allocatables**

Compile only. Default: `--dal`

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

### --*[*n*]*dbl

**Double**

Compile only. Default: `--ndbl`

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

### --*[*n*]*f95

**Fortran 95 Conformance**

Compile only. Default: `--nf95`

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

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

### --file *filename*

**Filename**

Compile and link. Default: not present

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

### --*[*n*]*fix

**Fixed Source Form**

Compile only. Default: not present

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

**Example**

```
        lf95 @bob.rsp bill.f90
```

If the command file `bob.rsp` contains --fix, then `bill.f90` will be interpreted as fixed source form even though it has the free source form extension `.f90`.

Specifying neither --fix nor --nfix will cause LF95 to interpret the source form according to the file's extension (see *"Filenames and Extensions"* on page 8). LF95 will not compile files (including `INCLUDE` files) containing both fixed and free source form in the same file.

### -g

**Debug**

Compile and link. Default: `-g` not present

Specify -g to instruct the compiler to generate an expanded symbol table and other information for the debugger. -g automatically overrides any optimization or parallelization option and forces -O0, no optimizations, so your executable will run more slowly than if optimization were used. -g is required to use the debugger.

### --help

**Display Compiler Options and Syntax**

Compile or link. Default: not present

Specifying this option alone on the command line will cause LF95 to print a summary of command-line options and syntax to the standard output and then exit.

### -I *dir*

**--include *dir*[:*dir1*[:*dir2* ...]]**

**Include Path**

Compile only. Default: current directory

Specify -I *dir* or --include *dir* to instruct the compiler to search the specified directory(ies) for Fortran include files. Multiple directories may be specified for --include with a colon-separated list of paths, which will be searched in the order specified. Note that -I will also affect module searches (see the Module Path option, *"-M dir"* on page 23 for directions on specifying module search paths). The current directory is always searched.

**Example**

```
        lf95 demo.f90 --include ../dir2/includes:../dir3/includes
```

In this example, the compiler first searches the current directory, then searches `..\dir2\includes` and finally `..\dir3\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, including suggestions on areas of possible improvement for your code and information on steps taken by the compiler for optimization and parallelization. --nwarn forces --ninfo.

### -l (lower-case L) *name*

#### Specify Library File

Link only. Default: `none.`

Specify a library file whose name is of the form `lib`*name*`.a` or `lib`*name*`.so`. Multiple library files may be specified with multiple -l options. Libraries are searched in the order that they appear on the command line (See “Linking Rules” on page 32.) This option and its argument are passed directly to the linker.

### -L *path*

#### Library Search Path

Link only. Default: LD_LIBRARY_PATH variable.

The -L option adds *path* to the list of directories that the linker searches for libraries, i.e., files beginning with “`lib`” and having the extension `.a` or `.so`. Note: if “`.`” (current directory) is not specified in your LD_LIBRARY_PATH variable, then you must specify `-L.` on the command line to search for files in the current directory. This option and its argument are passed directly to the linker.

#### Example

The following command line links main.o with libmine.a and libyours.so (residing in adjacent directories mylibs and yourlibs, respectively):

```
lf95 main.o -L../mylibs -lmine -L../yourlibs -lyours
```

Remember that, by default, the linker searches for shared libraries first.

### --*[n]*li

#### Recognize Lahey intrinsic procedures

Compile and link. Default: --li

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

## **--*[*n*]*long**

**Long Integers**

Compile only. Default: `--nlong`

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

## **--*[*n*]*lst *[ [spec=sval[, spec=sval]] ]***

**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 filenames consist of the basename of the first source filename plus the extension "`.lst`", placed in the current working directory (use `f=`*sval* suboption to override -- see below). The page width of the listing file is 274 columns, and no page breaks or additional headers are inserted into the body of the listing. Note that --nlst is overridden by --xref.

**Syntax**

--*[*n*]*lst *[*[*spec*=*sval[*, *spec*=*sval]*]*]*

**Where:**

*spec* is `f` for the listing filename, or `i` to include INCLUDE files. Each suboption must be separated by a comma and space, and the entire list of suboptions must be enclosed in square brackets ("`[]`").

For `f=`*sval*, the listing filename, *sval* specifies the listing filename to use instead of the default. If a file with this name already exists, it is overwritten. If the user specifies a listing filename and more than one source file then the driver diagnoses the error and aborts.

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

**Example**

```
lf95 myprog.f90 --lst [i=y]
```

creates the listing file myprog.lst and lists the include files.

**See also**

--*[*n*]*xref

## **--*[*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.

### --ml *target*

**Mixed Language**

Compile only. Default: not present

The --ml option is sometimes needed if your code calls or is called by code written in another language. The value of *target* will only affect procedures declared with the ML_EXTERNAL statement. Currently the only supported value for *target* is `cdecl`, which is needed for making calls to the system kernel. The --ml option is not needed for interfacing with g77 programs. See *"Mixed Language Programming"* on page 37 for more information.

### --mldefault *target*

**Mixed Language Default**

Compile only. Default: `-mldefault`

Specify the --mldefault options to set the default target language name decoration/calling convention for all program units. `--mldefault` affects name mangling for routine names in ML_EXTERNAL statements. Currently the only supported value for *target* is `cdecl`, which is needed for making calls to the system kernel. The --mldefault option is not needed for interfacing with g77 programs. See *"Mixed Language Programming"* on page 37 for more information.

### -M *dir*

**--mod *dir[:dir1[:dir2 ...]]***

**Module Path**

Compile only. Default: current directory

Specify --M *dir* to instruct the compiler to search the specified directory for LF95 module (`.mod`) files. Multiple directories may be specified using the -I option for each additional search directory. The directory specified by -M is searched first, current working directory is searched next, and the directories specified with -I are searched last.

Specify --mod *dir*... to instruct the compiler to search the specified directory or directories for LF95 module files. When using --mod, multiple directories may be specified using a colon separated list of directories. If multiple directories are specified, the first directory in the list is searched first, the current working directory is searched next, the remaining directories are then searched in order of appearance.

-M and --mod should not be used in combination on the same command line. When compiling procedures using modules, the path to all modules that are used either directly or indirectly must be specified. This also applies to modules that are already compiled.

When creating a new module, the .mod file will be placed in the directory specified with -M or the first directory specified by --mod. If the directory does not exist, the compiler will attempt to create it. If no directories are specified with -M or --mod, then module files are placed in the current working directory. Note that -I has no effect on module placement, even though it affects the order that directories are searched for existing modules. Module object (.o) files are placed in the current working directory. Note that any module *object* files created by previous compilations must be on the LF95 command line when linking.

**Example**

```
      lf95 modprog.f90 mod.o othermod.o -M ../mods -I ../other
```

or,

```
      lf95 modprog.f90 mod.o othermod.o --mod ../mods:../other
```

In these examples, the compiler first searches for module files in `../mods`, then searches the current working directory, and finally searches `../other.` All module files produced from `modprog.f90` are placed in the directory `../mods`. All object files produced by modprog.f90 are placed in the current working directory.

## { -O0 | -O }

### { --o0 | --o1 | --o2 }

**Optimization Level**

Compile only. Default: `-O`

Specify -O0 to disable optimization. -O0 is forced when the -g, --chk, or --chkglobal option is specified. See “-g” on page 20.

Specify -O to optimize for execution speed. To see details of steps taken by the compiler for optimization, specify the --info option. See “--[n]info” on page 21.

Specify --o2 to perform additional optimizations. Use of the --o2 option may significantly impact compilation speed.

-O0 and --o0 are equivalent.

-O and --o1 are equivalent.

## -o *name*

### --out *name*

**Output Filename**

Compile: Default is root name of source file, with extension `.o`

Link: Default is `a.out`, in current working directory

When not linking (i.e., when -c is specified), specify -o to override the default object filename and path. The default path is the current working directory. When linking (-nc specified or -c not specified), specify -o to override the output executable or library default filename. By default it is placed in the current working directory.

--out differs from the ld option -o in that LF95 uses --out to determine if a library is being built. -o is passed directly to ld. If the desired output is a library, use --out and specify an extension of .a or .so. See *“Shared Libraries”* and *“Archive Libraries”* on page 12.

**Example**

```
lf95 hello.f90 -c -o/home/mydir/hello.o
lf95 main.o --out maintest
```

## --*[n]*ocl (PRO version only)

**Process optimization control lines**

Compile only. Default: `--nocl`

--ocl causes optimization control lines (OCLs) to be processed. See *"Optimization Control Line"* on page 77 for more information.

## --*[n]*openmp (PRO version only)

**Process OpenMP directives.**

Compile and link. Default: `--nopenmp`

--openmp causes the compiler to process OpenMP directives in Fortran code. See *"OpenMP"* on page 89 for more information.

## --*[n]*parallel (PRO version only)

**Attempt automatic parallelization.**

Compile and link. Default: `--nparallel`

--parallel forces -O (full optimization). Note that the --parallel is ignored if the -g, --chk, or --chkglobal option is specified. To see the compiler's parallelization decisions, specify --info. See *"Overview of Multi-Processing"* on page 69 for more information.

## --*[n]*pca

**Protect Constant Arguments**

Compile only. Default: `--npca`

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

**Example**

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

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

## --*[n]*prefetch *[level]*

**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. *Level* must be either 1 or 2. 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 options will be ignored if --O0 or -g are used.

## --*[*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

### Quad Precision

Compile only. Default: `--nquad`

Specify --quad to extend all double-precision REAL and double-precision COMPLEX variables, arrays, constants, and functions to REAL (KIND=16) and COMPLEX (KIND=16) respectively. Default (single-precision) REAL entities remain unaffected. 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. Note that specifying -dbl -quad will not raise single-precision entities to quad precision.

## --*[*n*]*quiet

### Quiet Compilation

Compile only. Default: `--quiet`

Specifying --quiet suppresses the reporting of current file and program unit being compiled. Instead, only errors, warnings (with --warn), and informational messages (with --info) are displayed.

## --*[*n*]*sav

### SAVE Local Variables

Compile only. Default: `--nsav`

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

## --*[*n*]*shared

**Create Shared Library**

Link only.  Default: `--nshared`

Specify --shared to create a shared library rather than an archive (static) library (for more information, see *"Shared Libraries"* on page 12).

## --*[*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.

## --static

**Static System Runtime Libraries**

Link only.  Default: not present

Specify --static to create an executable linked only with static versions of libraries.  This is a GNU linker option.  For more information, see the man or info pages for ld, the GNU linker.

## --*[*n*]*staticlink

**Static Fortran Runtime Libraries**

Link only.  Default: `--nstaticlink`

Specify --staticlink to create an executable linked with the static LF95 Fortran runtime libraries, and the shared versions of the Linux system libraries.  Specifying --staticlink will result in a larger executable, because it does not depend on the presence of any Fortran runtime shared libraries.  (see *"Distributing LF95 Applications"* on page 33).

## --*[*n*]*swm *msg[,msg[,...]]*

**Suppress Warning Message(s)**

Compile only.  Default: `--nswm`

To suppress a particular warning or informational message that appears during compilation, specify its four digit number *msg* after --swm.  Multiple messages may be specified as a comma-separated list with no spaces.

**Example**

```
        --swm 1040,2005
```

This example would suppress warning messages `1040` and `2005`.  To suppress all warnings and informational messages, use --nwarn.  A list of warning and error numbers is in the file `RTERRMSG`.

## { --t4 | --tp | --tpp | --tp4 }

**Target Processor**

Compile only.  Default: `--tp`

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

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

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

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.

## --threads *N* (PRO version only)

**Number of threads**

Compile only. Default: the number of active processors on the system.

--threads specifies the number of instances (threads) to be created in the range $2 \leq N \leq$ number of CPUs active at runtime. If this option is specified, it eliminates the need for the compiler to produce overhead code identifying how many CPUs are available at execution time. It is also useful if there is a natural division of the problem into parallel segments and the number of segments is different from the number of available CPUs.

Be sure that the environment variable PARALLEL is set to the specified number (*N*) at runtime. The executable program that is generated by specifying this option is always executed with *N* CPUs, even if the program is moved to a machine with a different number of CPUs.

--threads requires --parallel. -g, --chk, or --chkglobal cause --threads to be ignored.

## --threadstack *N* (PRO version only)

**Thread Stack Size**

Compile only. Default: the executable stack size.

--threadstack sets the size of the stack for each thread to *N* kilobytes, where N is between 16 and 2048, inclusive. The maximum stack size for a Linux thread is 2048 kilobytes. This option takes precedence over the environment variable THREAD_STACK_SIZE (see *"Environment Variable THREAD_STACK_SIZE"* on page 72).

--threadstack requires --openmp or --parallel and must be specified for the file with the main program unit. -g, --chk, or --chkglobal cause --threadstack to be ignored.

## --threadheap *[size]* (PRO version only)

**Thread Heap Size**

Compile only. Default: 4096 bytes

If the --threadheap option is specified, local arrays in a procedure or parallel region that are larger than *size* bytes are allocated on the heap except for the following arrays:

- equivalenced arrays
- arrays that are namelist object

- arrays of derived type that specify default initialization
- arrays in common that have the PRIVATE attribute

*size* must be a positive number less than 2147483648. If the =*size* is omitted, 4096 is selected for *size*.

Execution performance may degrade when --threadheap is specified. Use this option only when the required thread stack size exceeds 2048 bytes.

--threadheap requires --openmp. -g, --chk, or --chkglobal cause --threadheap to be ignored.

## --*[*n*]*trace

**Location and Call Traceback for Runtime Errors**

Compile and link. Default: `--trace`

The --trace option causes a call traceback with routine names and line numbers to be generated with runtime error messages. With --ntrace no line numbers are generated. --trace might cause your program to run more slowly.

## --*[*n*]*trap

**Trap numeric exceptions**

Compile only. Default: `--ntrap`

The --trap option causes the Fortran runtime library to publish an error message on a divide by zero or overflow exception. The application then terminates. If the -WI,-i runtime option is specified (see *"Interrupt Processing"* on page 121), then no trapping occurs for overflow exceptions. If the -WI,-u runtime option is specified, then underflow exceptions are trapped (see *"Underflow Interrupt Processing"* on page 123).

## --*[*n*]*unroll *limit*

Compile only. Default: `--nunroll`

**Loop unrolling**

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

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

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

Note that -O forces --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 in the bss segment.

*size* is the minimum variable size that will be placed on the heap. Variables smaller than *size* are not placed on the heap.

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

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

### --*[*n*]*verbose

Compile only. Default: `--nverbose`

#### Verbose Output

Specify --verbose to see details of commands passed to all component tools used in the creation of object files, executable files, and libraries.

### --*[*n*]*version

#### Print Version Information

Compile and link. Default: `--nversion`

Specify --version to display product serial number, copyright, and version information when compiling or linking.

### --*[*n*]*warn

#### Warn

Compile only. Default: `--warn`

Specify --warn to display warnings at compile time. Note that --nwarn forces --ninfo.

### --*[*n*]*wide

#### Wide-Format Source Code

Compile only. Default: `--nwide`

Specify --wide to compile fixed-format source code that extends out to column 255. This option has no effect when compiling free-format source.

### --*[*n*]*wisk (PRO version only)

#### *Winteracter* Starter Kit

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

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

### --*[*n*]*wo

#### Warn Obsolescent

Compile only. Default: `--nwo`

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

### -x *arg*

#### Inline Code

Compile only.  Default:  do not inline

Specify -x to cause procedures to be inserted inline at the point they are referenced in the calling code.  Multiple arguments are separated by commas. At least one argument must be present.  The -x option may only be specified once per compile session.

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

If *arg* is a number with the letter “K” appended, arrays which have a size less than *arg* kilobytes are inlined.  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 the dash character “-”, all procedures having fewer than 30 lines of code and all local data are inlined.  If the dash argument is specified, no other arguments may be present.

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

### --*[*n*]*xref

#### Cross-Reference Listing

Compile only.  Default:  `--nxref`

Specify --xref to generate cross-reference information in the listing file.  By default, cross reference filenames consist of the basename of the source filename, plus the extension “`.lst`”, placed in the current working directory (see *“--[n]lst [ [spec=sval[, spec=sval]] ]”* on page 22).  Specifying --xref will override --nlst.

#### See also

--*[*n*]*lst

### --*[*n*]*zfm

#### Enable zero flash mode for SSE2 instructions

Compile only.  Default:  `--nzfm`

Specify --zfm enable zero flash 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 Rules

During the link phase, the driver submits object files and object file libraries to the linker for creation of the executable (or shared library) output file.

### Fortran 95 Modules

If your program uses Fortran 95 modules that have already been compiled, you must add the module object filenames (i.e., the source filename with extension `.o`) to the LF95 command line when linking. Compiling a Fortran 95 module will generate an object (`.o`) file and a module (`.mod`) file if the source file contains executable code. If the source file does not contain any executable code but does contain public entities, then only a `.mod` file will be generated.

### How the Linker Finds Libraries

The linker reads individual object files and object module libraries, resolves references to external symbols, and writes out a single executable file (or shared library).

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

1. in any directories specified with the `-L` option.
2. in any directories specified by the `LD_LIBRARY_PATH` environment variable.

**Note:** the current working directory "`.`" will not be searched unless it is specified by the `-L` option or the `LD_LIBRARY_PATH` environment variable.

In each case, the linker will first attempt to locate a shared library (with a `.so` file extension) containing the desired symbol(s). If that is not found, then it will seek an archive or static library (with a `.a` file extension). The `--staticlink` option does not affect this behavior; this option only determines the specific group of runtime libraries that will be linked to the executable.

Searching rules for INCLUDE files and Fortran 95 modules are governed by the compiler, not the linker. See *"-I dir"* on page 20 and *"-M dir"* on page 23 for discussion.

### Object File Processing Rules

Object files are processed in the order they appear on the command line.

### How the Linker Selects Objects

The `ld` linker applies the following rules when searching object libraries:

1. Any libraries specified using the `-l` option are searched in the order in which they appear in the LF95 command string before the LF95 runtime library, or any libraries appearing in directories specified by the `-L` option or the `LD_LIBRARY_PATH` environment variable. The compiler writes the default LF95 library names into each object file it generates.
2. Each library is searched until all possible external references are resolved. If necessary, system libraries appearing in `/lib` or `/usr/lib` may also be searched.

### Linker Options

In most cases, LF95 passes unrecognized options on to the linker; however, some linker options may conflict with existing LF95 options. In this case, an option may be passed directly to the linker from the LF95 command line using the `-Wl` option. This option behaves as documented in the man pages for `gcc`, the GNU C compiler (coincidentally, -`Wl` is the same option used to indicate runtime options as described in Appendix B, *Runtime Options*).

For further information, see the man pages for `ld`, the GNU linker.

## Distributing LF95 Applications

When you distribute applications built with LF95, you need to be aware of the shared (dynamic) libraries that your application requires to run on the target platform. You can use the Linux command `ldd` to display the shared libraries required by your application.

Any shared libraries that have been created must be distributed them with your application.

You must link with the --staticlink option, which will bind the LF95 Fortran static runtime libraries to the executable (see *"--[n]staticlink"* on page 27). You are not allowed to distribute the LF95 Fortran shared libraries (`*.so.1`) residing in the `lib` subdirectory of your LF95 installation.

The remaining required shared libraries (usually residing under the `/lib` directory) are the GNU C runtime libraries which will be available on any Linux system that has glibc installed. Distributing these libraries is not recommended and is governed by a GNU Public License. These shared libraries allow your application to use the GNU C runtime of the target Linux system, whether it be newer or older. Note that a program built on a system running a newer version of glibc might not execute properly on a system running an older version. It is recommended that you build your application on the earliest version available for best portability.

If it is necessary for you to statically link the GNU C runtime libraries with your application, you must link with the -static linker option. Your distribution will be governed by a GNU Public License and the Lahey Software License Agreement, which states:

“If you distribute User Programs that statically link the Lahey/Fujitsu Fortran and the GNU C runtime libraries into your program, you may redistribute the Lahey/Fujitsu Fortran static libraries (`*.a`) and the `fj90rt0.o` file with your programs for the sole purpose of allowing your customers to rebuild the programs you distribute, provided you instruct your customers, and they agree, to remove the Lahey/Fujitsu Fortran static libraries (`*.a`) and the `fj90rt0.o` file from their computer systems after rebuilding the programs you distribute.”

## 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, Linux, 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. A complete set of demonstration programs may be downloaded from the Lahey web site. The f90gl interface was developed by William F. Mitchell of the Mathematical and Computational Sciences Division, National Institute of Standards and Technology, Gaithersburg, MD, in the USA. For information on f90gl, see the f90gl web page at `http://math.nist.gov/f90gl`. For more information on using OpenGL and f90gl with LF95, see the HTML help file “wisk.htm” in the help directory provided with LF95 PRO.

## Scientific Software Libraries (PRO Version only)

The Scientific Software Libraries (SSL2) are a library of subroutines and functions designed to aid in the solution of common scientific and engineering problems. Three versions of the library are provided, a generic version suitable for use with any supported processor, a multithreaded version suitable for use with any supported multiple processor hardware, and a highly optimized multithreaded version for use with systems using multiple Pentium 4 processors. For more information concerning the SSL2 libraries or specific procedures, see the SSL2 PDF documents in the manuals directory of your LF95 distribution, or consult the man page for the procedure in question.

## BLAS and LAPACK Libraries (PRO Version only)

Multithreaded versions of the BLAS and LAPACK libraries are provided. These libraries provide a standardized set of procedures for solving linear algebra and matrix algebra problems. Two versions of the BLAS library are provided, a multithreaded version suitable for use with any supported multiple processor hardware, and a highly optimized multithreaded version for use with systems using multiple Pentium 4 processors. The LAPACK library is only supplied in a multithreaded version, but may be linked with the Pentium 4 optimized versions of BLAS. For more information concerning the BLAS and LAPACK libraries or specific procedures, see the BLAS/LAPACK PDF document in the manuals directory of your LF95 distribution.

## Porting Code Between Windows and Linux

If your code is F77, F90, or F95 standard conforming, it will port to Linux simply by recompiling. If you are using the Winteracter or Gino GUI libraries, you can recompile your code and link with the Linux version of these libraries without having to make any other changes. If you are using Automake, the basic structure of the automake.fig configuration file will remain the same. If any code or data contains path information, you will have to change the Windows directory separator "\" to the Unix separator "/", and make sure that pertinent files are in the indicated directories. If you use environment variables, you will need to convert from Windows style "%var%" to Unix style "$var". Many non standard extensions are supported under both the Windows and Linux environments. If an extension is not supported, it will most likely cause an "undefined symbol" error when linking. If your code uses the SYSTEM subroutine, you should consult your Language reference. Although the basic form of the SYSTEM command is supported under both systems, optional arguments are not supported on the Linux side.

## 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 --ng -O --npca --nsav --ntrace
```

Also, use `--t4, --tp, --tpp, or tp4` depending on your preferred target processor. Note that code compiled with `--tpp` will only run on Pentium Pro or newer compatible chips. Note that code compiled with `--tp4` will only run on Pentium 4 or newer compatible chips.

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 -x 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 (-O) 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

LF95 code can call and be called by code written in certain other languages. With LF95 one can create object and library files for use with the language systems in the table below. Calls can be made from Fortran to Fortran, from Fortran to another language, and from another language to Fortran. If you are calling LF95 routines from a language system other than LF95, it may be necessary to refer to that language system's documentation for more information.

## What Is Supported

Lahey/Fujitsu Fortran 95 supports mixed language interfaces to the following languages and operating systems (this list is subject to change -- see `READ_ML` for any changes):

**Table 4: Compiler Support for Mixed Language**

| Language System | --ml option (see below) |
|---|---|
| Linux kernel and standard C libraries | --ml cdecl |
| Gnu C | --ml cdecl |
| Fujitsu C | --ml cdecl |
| Gnu Fortran77 | (none) |

## Declaring Your Procedures

In order to reference a procedure across a mixed language interface, the LF95 compiler must be informed of the procedure name and told how to "decorate" this name as it appears in the object file. These procedure names are defined with the `ML_EXTERNAL` statement (see "*ML_EXTERNAL Statement*" in the LF95 Language Reference). The `DLL_EXPORT` and

DLL_IMPORT statements used in the LF95 Windows product are still supported, but their effect is identical to ML_EXTERNAL since the calling conventions are the same for Linux static and shared libraries.

Please note that in general, mixed language procedure names are *case sensitive* (unlike the Fortran naming convention, which ignores case). ML_EXTERNAL is used when defining a Fortran procedure and when referencing an external procedure. The type of mixed language interface 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 procedures from multiple languages you can do so by putting the references in separate source files and compiling them separately.

The table below describes the varieties of procedures that may be found in an LF95 program, along with the form taken by the procedure's default external name (i.e., the name seen by the linker). Procedures MAIN__() and main() play a special role in mixed-language programs. This is described in *"Program Control: main() and MAIN__()"* on page 50.

**Table 5: Default External Names for Fortran Procedures**

| Procedure Name | Seen from outside as: |
|---|---|
| FUNCTION MyFunc()<br>SUBROUTINE MySub() | myfunc_<br>mysub_ |
| intrinsic procedure proc1() | f_proc1<br>or<br>g_proc1 |
| main program | MAIN__ |
| Fortran startup/initialization routine | main |
| common block a | a_ |

The external names of Fortran functions and subroutines may be modified by using the `ML_EXTERNAL` statement, along with the --ml compiler option. The purpose of the `ML_EXTERNAL` statement is to modify the “name decoration” or “name mangling” that is applied to the external procedure name (in accordance with the `--ml` compiler option) and to allow case to be preserved.

**Table 6: Effect of --ml Option on External name of Fortran Procedure MySub1(), Declared as ML_EXTERNAL**

| --ml option | Seen from outside as: |
|---|---|
| `--ml cdecl` | `MySub1` |
| `--ml not specified` | `MySub1_` |
| not declared as `ML_EXTERNAL` | `mysub1_` (`--ml` has no effect) |

Note that if `MySub1()` is not declared as `ML_EXTERNAL`, then the `--ml` option has no effect, and its external name will always be `mysub1_`. Fortran naming conventions can be accommodated in C by declaring the C function as lower case and adding a trailing underscore character, thus eliminating the need for the `ML_EXTERNAL` statement or the `--ml` compiler option. On the other hand, if Fortran is calling a C library for which no source code is available, then the `ML_EXTERNAL` statement and the `--ml` compiler option are required.

## Interfacing with g77 (GNU Fortran)

When writing procedures in LF95 that will call or be called from g77, it is not necessary to specify the --ml option on the command line or apply the ML_EXTERNAL statement to the procedure name. It is, however, important to link to the proper libraries so that intrinsic procedures may be resolved. See the `examples/mix_lang` directory under your installation root directory for examples of how to link with g77 objects.

## Interfacing with Non-Fortran Languages

When you create a Fortran library or object file, you will usually indicate each procedure that you want made available using the `ML_EXTERNAL` statement. The procedure may be a subroutine or a function. When a Fortran function returns a value, the calling language must match the value to its corresponding data type as described in Table 8 on page 43.

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

When you create a Fortran program that references non-Fortran procedure(s), you declare the non-Fortran procedure name(s) with the `ML_EXTERNAL` statement in your Fortran code. The syntax of the `ML_EXTERNAL` statement in this case is:

`ML_EXTERNAL` *external-name-list*

where *external-name-list* is a comma-separated list of names of procedures referenced in this scoping unit. The procedures may be subroutines or functions. Non-Fortran functions may only return data types specified in Table 7 on page 41.

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

These codes must be compiled using LF95's `--ml` *target* option in order to be callable by language *target* (See "--ml target" on page 23.).

Note that `ML_EXTERNAL` is a statement and not an attribute. In other words, `ML_EXTERNAL` may not appear in an attribute list in an INTEGER, REAL, COMPLEX, LOGICAL, CHARACTER or TYPE statement.

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

## Passing Data

Data may be passed to or from other language systems as arguments, function results, external (COMMON) variables, or in files. LF95 does not support arrays of pointers passed from C, or pointers with more than one level of indirection.

LF95's calling conventions are as follows:

- All arguments are pass-by-address, not pass-by-value as in C. LF95 can pass arguments by value to other languages, using the VAL() intrinsic.
- Arrays of pointers cannot be passed from C to Fortran.
- COMPLEX and derived type arguments can be passed as pointers to structures. Because C does not have a native type for complex data, it must be declared as a structure. For example, Fortran default COMPLEX is declared in C as
```
struct {
float real;
float imaginary;
} complex;
```
- When passing data via a file, the file must be closed prior to calling the non-Fortran procedure.
- Fortran common blocks can be accessed as an external or "global" structure from C. For example, the named common block,

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

can be accessed as

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

"Blank" (unnamed) common is treated the same way; the structure is named `_BLNK_` instead of `my_common_`.

Data passed between Fortran and C programs must have corresponding attributes. The following table describes corresponding data types between C and Fortran. Note that some of the listed data types will be unavailable on some C compilers.

**Table 7: Corresponding Data Types in Fortran and C**

| Data Type | Fortran | C | Comments |
|---|---|---|---|
| one-byte logical | `LOGICAL(1) L1` | `char L1;` | 1 byte |
| two-byte logical | `LOGICAL(2) L2` | `short int L2;` | 2 bytes |
| four-byte logical | `LOGICAL(4) L4` | `long int L4;` | 4 bytes |
| eight-byte logical | `LOGICAL(8) L8` | `long long int L8;` | 8 bytes |
| one-byte integer | `INTEGER(1) I1` | `signed char I1;` | 1 byte |
| two-byte integer | `INTEGER(2) I2` | `short int I2;` | 2 bytes |
| four-byte integer | `INTEGER(4) I4` | `long int I4;` | 4 bytes |

**Table 7: Corresponding Data Types in Fortran and C**

| Data Type | Fortran | C | Comments |
|---|---|---|---|
| eight-byte integer | `INTEGER(8) I8` | `long long int I8;` | 8 bytes |
| real | `REAL(4) R4` | `float R4;` | 4 bytes |
| double-precision real | `REAL(8) R8` | `double R8;` | 8 bytes |
| quadruple-precision real | `REAL(16) R16` | `long double R16;` | 16 bytes |
| complex | `COMPLEX(4) C8` | `struct {float r, i;} C8;` | 8 bytes |
| double-precision complex | `COMPLEX(8) C16` | `struct {double r, i;} C16;` | 16 bytes |
| quad-precision complex | `COMPLEX(16)C32` | `struct {long double r, i;} C32;` | 32 bytes |
| character (fixed length) | `CHARACTER*10 S` | `char S[10]` | See examples for assumed-length |
| derived type | `TYPE TAG`<br>`SEQUENCE`<br>`INTEGER I4`<br>`REAL(8) R8`<br>`END TYPE`<br>`TYPE(TAG) D` | `struct tag`<br>`{`<br>`int I4;`<br>`double R8;`<br>`} D;` | Size (in bytes) = sum of all components |
| array of pointers | `not allowed` | `*myarray[10]`<br>`**hisarray` | |

## Returning Function Values to C

Fortran functions are called from C as functions returning a value, with all arguments passed by reference. Values are passed on the stack, with the exception of `COMPLEX` and `CHARACTER` data, in which case the values are passed via the argument list. The following table lists the data types that may be returned to C from a Fortran function. In the third column of the table (“examples” column), the variable `result` represents the value returned by the Fortran function `myfunc()`. In the last example, the variable `strlen` represents the length of the character value returned by `myfunc()`.

This section does not discuss Fortran subroutines, which are called from C as "void" functions. This concept is illustrated in a later section, *"Passing and Receiving Arguments"* on page 45.

**Table 8: Declaring C Result Types for Fortran Function Types**

| Fortran Function Type | C Result Type | Example |
|---|---|---|
| `INTEGER(1)` | `signed char` | `result = myfunc_();` |
| `INTEGER(2)` | `short int` | `result = myfunc_();` |
| `INTEGER(4)` | `long int` | `result = myfunc_();` |
| `INTEGER(8)` | `long long int` | `result = myfunc_();` |
| `LOGICAL(1)` | `unsigned char` | `result = myfunc_();` |
| `LOGICAL(2)` | `short int` | `result = myfunc_();` |
| `LOGICAL(4)` | `long int` | `result = myfunc_();` |
| `LOGICAL(8)` | `long long int` | `result = myfunc_();` |
| `REAL(4)` | `float` | `result = myfunc_();` |
| `REAL(8)` | `double` | `result = myfunc_();` |
| `REAL(16)` | `long double` | `result = myfunc_();` |
| `COMPLEX(4)` | `void` | `myfunc_(&result);` |
| `COMPLEX(8)` | `void` | `myfunc_(&result);` |
| `COMPLEX(16)` | `void` | `myfunc_(&result);` |
| `CHARACTER(LEN=*)` | `void` | `myfunc_(&result,len);` |
| Derived Type | not applicable | not applicable |

For example, the Fortran function:

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

corresponds to the C prototype:

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

To illustrate returning an assumed-length character value, the Fortran function:

```
function cfun()
character(len=*) :: cfun
cfun = ‘1234567890’
end function cfun
```

is invoked from C as follows:

```
void cfun_(char *str1, int strlen);
MAIN__()
{
  char mystr[10];
  cfun_(mystr,10);
}
```

The preceding example may be a bit confusing, since it runs counter to the intuitive concept of a function returning a value. For further explanation, see *“Passing Character Data”* on page 47.

## Returning Function Values to Fortran

C functions are also called by Fortran as functions returning a value. By default, all arguments are passed to C by reference. Arguments may also be passed to C by value using LF95’s VAL() intrinsic. It is not possible to return character strings or structures from C.

Fortran calls “void” C functions in the same manner that it calls Fortran subroutines. This concept is illustrated in the section below, *“Passing and Receiving Arguments”* on page 45.

**Table 9: Declaring Fortran Result Types for C Function Types**

| C Function Type | Fortran Result Type | Example |
|---|---|---|
| `void` | not applicable | `call my_c_func()` |
| `signed char` | `INTEGER(1)` | `result = my_c_func()` |
| `short int` | `INTEGER(2)` | `result = my_c_func()` |
| `long int` | `INTEGER(4)`<br>`LOGICAL(4)` | `result = my_c_func()` |
| `long long int` | `INTEGER(8)` | `result = my_c_func()` |
| `float` | `REAL(4)` | `result = my_c_func()` |
| `double` | `REAL(8)` | `result = my_c_func()` |
| `long double` | `REAL(16)` | `result = my_c_func()` |
| `char` | cannot be accepted | not applicable |
| `structure` | cannot be accepted | not applicable |

## Passing and Receiving Arguments

By default, Fortran passes arguments “by reference” (i.e., it passes the address of each variable in the argument list, rather than the value of the argument, on the program stack); however, many C functions expect variables to be passed “by value” on the program stack. This practice can be accommodated by applying the VAL() intrinsic to the variable as it appears in the argument list of the Fortran reference to the function.

In all subsequent C code examples, a declaration of `int` is synonymous with `long int`. Note that any array arguments or arguments of type COMPLEX must not be passed by value to C; they should always be passed by reference. Character data is a special case -- it may be passed using either the CARG intrinsic or `VAL(OFFSET())`. See the section below, *“Passing Character Data”* on page 47 for further illustration.

#### Example: Passing Arguments by Value from Fortran to C

The C function

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

is called from Fortran as follows:

```
integer i, j, k
j = 3
k = 4
call mysum(i, val(j), val(k))
write (*,*) ‘ Result: j+k = ‘, i
```

#### Example: Passing Arguments by Reference from C to Fortran

Variables can be passed by reference from C using the l-value operator (&). The Fortran function

```
integer function myfunc(x, y)
integer x, y
myfunc = x + y
return
end function
```

is called from C as

```
MAIN__()
{
  long int myfunc_(*long int i, *long int j);
  long int i, j, k;
  i = 5
  j = 7
  k = myfunc_(&i, &j)
}
```

## Passing Arrays

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

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

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

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

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

The equivalent C code would look like:

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

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

## Passing Character Data

Character arguments are passed as pointers to strings. When a Fortran program unit contains character dummy arguments, then any routine calling that program unit must append to the end of the argument list the length of each of the corresponding character actual arguments. The length must be passed by value, as a four-byte integer (`long int`), to Fortran.

For example, the Fortran subroutine:

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

corresponds to this prototype in C:

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

When passing a character string from Fortran to C, Fortran will by default append a “hidden” integer value, representing the length of the string, to the end of the argument list. This integer is passed by value. If more than one character string is passed, the length values appear in the same order as the strings, at the end of the argument list. To prevent the length value from being added, apply the CARG() intrinsic or combine the VAL(OFFSET()) intrinsics, so that only the pointer to the string is passed.

In addition, C requires a NULL terminator (i.e., CHAR(0), a byte whose value is zero) at the end of a character string in order to process it. LF95 does not supply this; hence it must be appended to a character literal or character variable before it is passed to C. Furthermore, Fortran pads the end of the string with blanks to fill its entire declared length. If this padding is not desired then it must be removed by applying the TRIM() intrinsic and appending a NULL before the string is passed to C.

**Example: Passing Character Variables and Character Constants from Fortran to C**

The following Fortran program

```
program strtest
character*20 mystr
mystr = 'abcde'
call sub(mystr)
call sub('abcde'//char(0))
call sub2(carg(trim(mystr)//char(0)))
call sub2(val(offset(mystr)))
call sub2(carg('abcde'//char(0)))
end
```

and the following C subroutine

```
void sub_(str1,i)
char *str1;
long int i;
{
  printf("hidden length = %i\n",i);
  printf("%sHi!\n",str1);
}
void sub2_(str1)
char *str1;
{
printf("%sEnd.\n",str1);
}
```

produce the following output:

```
hidden length = 20
abcde               Hi!
hidden length = 6
abcdeHi!
abcdeEnd.
abcde               End.
abcdeEnd.
```

**Example: Passing String Variables from C to Fortran**

The following Fortran function has assumed-length character dummy arguments and returns an assumed-length character result:

```
function MYFUNC(str1, str2)
character(len=*) :: str1, str2, myfunc
myfunc = str1//str2//char(0)
return
end
```

When called by the following C program,

```
void myfunc_(char *str1, int i, char *str2, \
             char *str3, int j, int k);
MAIN__()
{
/* Leave space for NULL in character declarations */
char res[10], ch[4], msg[7];
strcpy(ch, "Hi ");
strcpy(msg, "there!");
myfunc_(res, 10, ch, msg, 3, 6);
printf("Result received by C: %s\n", res);
}
```

The following output is generated:

```
Result received by C:  Hi there!
```

In the call to `MYFUNC` from C, the first and second arguments are the value and length, respectively, of the result returned by `MYFUNC`. The last two arguments are the respective lengths of the character arguments being passed to `MYFUNC`.

## Passing Data through Common Blocks

The variables in a Fortran common block may be referenced as C structure members.

**Example: Named Common**

In the following Fortran program, the variables in common block “ext”

```
common /ext/ i, j
i = 1
j = 2
call sub()
end
```

are accessed by a C function as follows:

```
extern struct tab {
int i, j;
} ext_;
void sub_()
{
printf("i=%i j=%i\n", ext_.i, ext_.j);
}
```

### Example: Blank Common

Passing data via blank common is accomplished in the same manner as in the above example, except in the C code, the name `ext_` is replaced by `_BLNK_`.

## Program Control: main() and MAIN__()

If the top level of control in a mixed-language program resides in the non-Fortran language system (i.e., control is first passed to the non-Fortran portion of the program), the top-level procedure must be given the name `MAIN__()`. It must not be given the name `main()`, as this is reserved for startup and initialization of the Fortran runtime environment.

### Example: Passing Control First to a C Program

The following C program calls Fortran subroutine SUB() and then exits.

```
void sub_();
MAIN__()
{
sub_();
}
```

## Calling Standard C Libraries

When calling functions in the Linux kernel and standard C libraries, it is necessary to apply the ML_EXTERNAL statement to the function name, and compile with the --ml compiler option.

### Example: Calling a Linux Kernel Function

The following Fortran program illustrates a call to the standard function usleep().

```
program callsys
ml_external usleep
write(*,*) 'Going to sleep...'
!  sleep for 10 seconds
call usleep(10000000)
write (*,*) ' Wake up!'
end program
```

The above program must be compiled using the command line,

```
lf95 callsys.f90 --ml cdecl
```

# 4 Command-Line Debugging with fdb

fdb is a command-line symbolic source-level debugger for Fortran 95, C, and assembly programs.

Before debugging your program you must compile it using the `-g` option (see *"Compiler and Linker Options"* on page 14). The `-g` option creates additional symbolic debugging information within the executable code.

This chapter contains references to debugging of C code. These references are meant for C programs compiled with `fcc`, the Fujitsu C compiler. fdb is not compatible with the debug information generated by `gcc`, the GNU C compiler. It is, however, possible to debug LF95 programs using `gdb` (GNU debugger), subject to the following restrictions:

Fortran 90/95 specifications are not supported in gdb.

The contents of COMMON can only be examined in gdb by examining memory and interpreting the values there.

Fortran procedures must be specified as lowercase with trailing underscore (_). You can step through module procedures but you cannot set a breakpoint or examine the values of variables or parameters.

Fortran variables must be specified in capital letters.

## Starting fdb

To start `fdb` type:

`fdb` *[exefile] [corefile]*

**Where:** *exefile* is the name of an executable file compiled with the `-g` option, and *corefile* is the name of the core file (if any) produced by abnormal termination of the executable. If *exefile* is not supplied, then `fdb` will assume the executable file is `a.out`. If *corefile* is not supplied, then `fdb` will assume the core dump file is `core`.

If `core` is present in the current directory, or if *corefile* is specified, then `fdb` will start with the current line of code being the one that caused the abnormal termination, and the current file being the one that contains that line of code. If `core` or *corefile* is not a dump of *exefile,* then there will be no debug information available.

Otherwise, if no `core` file is available or *corefile* does not exist, then `fdb` starts with the current line of code being the first executable line of the file containing the main program.

## Communicating with fdb

### 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 65).

### Values

Numeric values can be of types integer, real, unsigned octal, or unsigned hexadecimal. Unsigned octal values must begin with a `0` and unsigned hexadecimal values must begin with `0x`. 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

When communicating with fdb, all procedure names must be in lower case, regardless of the case used in the source file. The main program name, when not specified in a PROGRAM statement, is `main`. In order to prevent user names from conflicting with intrinsic or runtime library names, the compiler "decorates" procedure and common block names by adding an underscore, '_', after the corresponding name specified in the Fortran source program. When referencing an external or module procedure or a common block in fdb, the trailing underscore is optional. However, when referencing any internal procedures, the name must be specified with the trailing underscore.

# 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***. Character literals must be enclosed by quotation marks (the symbol ", which is ascii 34). File names must be enclosed by the grave accent (the symbol `, which is ascii 96).

## 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. If single-stepping or other program control is desired, a breakpoint must be set before issuing the `run` command, otherwise the program will immediately run to completion. For an explanation of breakpoints, see *"Breakpoints"* on page 55. A breakpoint can also be set at `MAIN__`, the main Fortran entry point. Do not set a breakpoint at `main`; no debug information will exist there.

### Run

Executes the program without arguments. The “R” should be upper case. As explained above, a breakpoint must be set before issuing this command if single-stepping or other control is desired.

### kill
#### <ctl-c>

Forces cancellation of the program. <CTL-C> (control+c) has the same effect as the kill command.

### tty *dev*

Direct standard error I/O to device *dev* in the next run.

### 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

### setenv
#### show environment

All environment variables and their values are displayed.

### setenv "*var*"
#### show environment "*var*"

Environment variable *var* and its value are displayed

### setenv "*var*" "*s*"

The environment variable *var* is set to the value *strings*.

### unsetenv "var"

The variable *var* is deleted from the environment.

### quit

Ends the debugging session.

## Help Commands

**help**
Display the list of all commands

**help *cmd***
Display help for command *cmd*

**help "*regex*"**
Display help for all commands corresponding to regular expression *regex*. Note that the quotation marks (ascii 34) are required.

## Shell Commands

**cd *dir***
Change working directory to *dir*

**pwd**
Display the current working directory path

**sh *cmd***
Execute arbitrary shell command *cmd*

## 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. Note that the “apostrophes” used in ` *file* `, above, are the grave accent (ascii 96), not the standard apostrophe character.
- *proc* [+|- *offset*] specifies the line number corresponding to the entry point of function or subroutine *proc* plus or minus *offset* lines. When using this syntax, *proc* may not be a module or internal procedure.
- *proc@inproc* specifies internal procedure inproc within proc.
- [*mod@*]*proc*[*@inproc_*] specifies procedure *proc* contained in module

*mod* or internal procedure *inproc* within module procedure *proc*. Note that a breakpoint may be set on a module procedure without specifying the module name. If there is more than on module with a procedure of a given name, then you will be prompted to select from a list.

- **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, structure components, 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.)

### <u>b</u>reak [ *'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" used in `*file*`, above, are the grave accent (ascii 96), not the standard apostrophe character.

### <u>b</u>reak [ *'file'* ] *procname*

Sets a breakpoint at the entry point of the procedure *proc* in the source file *file*. If omitted, *file* defaults to the current file. Note that the "apostrophes" used in `*file*`, above, are the grave accent (ascii 96), not the standard apostrophe character.

### <u>b</u>reak **addr*

Sets a breakpoint at address `addr`.

### <u>b</u>reak

Sets a breakpoint at the current line.

### <u>b</u>reako<u>f</u>f [#*n*]

Disables breakpoint number *n*. When #*n* is omitted, all breakpoints are disabled. The breakpoints still exist and can be enabled using the breakon command. Note that the "#" symbol is required.

### <u>b</u>reako<u>n</u> [#*n*]

Enables breakpoint number *n*. When #*n* is omitted, all breakpoints are enabled. 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 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” used in `*file*`, above, are the grave accent (ascii 96), not the standard apostrophe character.

### **delete [ *‘file‘* ] *procname***

Removes the breakpoint for the entry point of the procedure *procname* in the source file *file*. If omitted, *file* defaults to the current file. Note that the “apostrophes” used in `*file*`, above, are the grave accent (ascii 96), not the standard apostrophe character.

### **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 procedure 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 *location***

Continues program execution until reaching the location *location*. The same syntax rules as for breakpoints apply.

**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 procedure that includes the current breakpoint.

**goto [ *‘file‘* ] *line***

Execution is restarted from the specified line *line* in file *file*.

### jump [ '*file*' ] *line*

Changes the program counter (jumps) to the address corresponding to the specified line *line* in file *file*.

### jump **addr*

Changes the program counter (jumps) to address *addr*.

## 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

Select stack frame number *n*. If *n* is omitted, the current stack frame is selected.

### 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 (see *"Registers"* on page 53 for register notation details).

### 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 (see *"Registers"* on page 53 for register notation details).

**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 (see *"Registers"* on page 53 for register notation details).

**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 -- see format descriptions in `print` command above).

*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 sigaround

Displays the last 5 lines and the next 5 lines, including the line of the current file nearest the address where the signal occurred.

### 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. Note that the “apostrophes” used in `*file*`, above, are the grave accent (ascii 96), not the standard apostrophe character.

### 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. Note that the “apostrophes” used in `*file*`, above, are the grave accent (ascii 96), not the standard apostrophe character.

### list [ func[tion ]] *procname*

Displays the last 5 lines and the next 5 lines of the entry point of the procedure *procname*.

### 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 procedure that contains address *addr1*.

### disas *procname*

Displays all instructions of the procedure *procname* 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. Refer to *“print [ [:F] variable [ = value ] ]”* on page 61 for an explanation of *F*.

### 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* (quotation marks required).  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* (quotation marks required).  When *regex* is omitted, all variable names and types are displayed.

## Scripts

**script *`script`***
The commands in file *script* are executed.  Note that the “apostrophes” used in `*script*`, above, are the grave accent (ascii 96), not the standard apostrophe character.

**alias *cmd*  "*cmd-str*"**
Assigns the fdb command(s) in *cmd-str* (quotation marks required) 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:

| | |
|---|---|
| `stop` | Execution stopped when signal sig encountered |
| `nostop` | 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 (quotation marks required). The initial (default) value is “`fdb*`”.

### param printarray on|off

When the value is “on,” the elements of arrays are displayed, one element per line, in response to the print command. The initial (default) value is “off,” which causes elements to be displayed as a comma-separated list which wraps around the end of the console screen.

### param printstructure on|off

When the value is “on,” the elements of derived types (structures) are displayed, one element per line, in response to the print command. The initial (default) value is “off.”

### 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.

### 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 can also be controlled via the `FDB_SRC_PATH` environment variable, which is comprised of a list of directories separated by colons.

### 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

### <u>b</u>reak<u>a</u>ll *mdl*

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

### <u>b</u>reak<u>a</u>ll *func*

Set a breakpoint in all internal procedures in procedure *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.

## Memory Leak Detection

### param leak off | mem | all

Controls level of memory leak checking, where the level is determined as follows:

| | |
|---|---|
| `off` | No leak checking (default). |
| `mem` | Memory manipulation functions and statements (such as ALLOCATE, DEALLOCATE, malloc, free, and memcpy) are checked. |
| `all` | Character string system procedures are checked, in addition to those checked by the `mem` option. |

#### param leak

Reports current level of leak checking.

#### show leak  log | error | summary

Displays memory leak information, where the type of information displayed is as follows:

| | |
|---|---|
| `log` | Displays procedures being monitored, in the order that they are called. |
| `error` | Displays error messages. |
| `summary` | Summary information only. |

### Processes and Threads

#### ps [*pid*]

Displays information about process-id *pid*.  If *pid* is not specified, then information is displayed for all process-id’s.

## Restrictions

1 An adjustable array that is a dummy argument cannot be debugged if it appears at the beginning of a procedure or a function in a Fortran program.

Example:

```
subroutine sub(x,y,i)
real x(5:i)
real y(i+3)
```

In this example, adjustable arrays "x" and "y" cannot be debugged at the subroutine statement.

2 The dummy argument of a main entry cannot be debugged at the sub-entry in a Fortran program.

Example:

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

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

3 Breakpoints cannot be set at any executable statements of an include file in Fortran programs.

4. An array of an assumed size can be debugged only for the lower bound of the array in a Fortran program.
5. The statement label in a Fortran program cannot be debugged.
6. In a Fortran program, the breakpoint you can make at the beginning of a procedure may vary in cases where the -g or --chk option is specified.
7. In include files that contain expressions in Fortran and C, programs cannot be debugged.
8. If you want to set a break point in the main procedure which has no program statement in a Fortran program, the break point is set at the first executable statement or declare statement.
9. When in the Fortran program the continue statement has no instruction, even if you want to set a break point at this statement, the break point is set at the next executable statement.

   Example:

   ```
      integer :: i
      assign 10
   10 continue
      i = 1
   ```

   A break point set at the continue statement will break at the next executable statement (i = 1).
10. The index name of the FORALL statement in a Fortran program cannot be debugged.
11. In Fortran, a name exceeding 2048 bytes cannot be displayed.
12. In Fortran, the value of floating-point registers cannot be displayed or set."

# 5 Multi-Processing (PRO version only)

This chapter describes the method of processing a Fortran program in parallel. Processing a Fortran program in parallel is called *multi-processing*.

## Overview of Multi-Processing

In this document, multi-processing means that one program is executed on two or more CPUs that can work independently and simultaneously. As used here, it does not mean executing two or more programs simultaneously. Consider the following code:

```
do i = 1, 50000
  a(i) = b(i) + c(i)
end do
```

Different iterations of the DO loop are executed on different CPUs at the same time.

CPU 1:

```
do i1 = 1, 25000
  a(i1) = b(i1) + c(i1)
end do
```

CPU 2:

```
do i2 = 25001, 50000
  a(i2) = b(i2) + c(i2)
end do
```

## Performance Improvement

The effect of multi-processing is to save elapsed execution time by using two or more CPUs simultaneously. For instance, if a DO loop can be executed in parallel by dividing it as shown above, then, theoretically, the execution time of this DO loop may be cut in half. In practice, improving performance requires some care and some work on the part of the programmer, as explained in the next section.

Although the elapsed time usually will be decreased by multi-processing, the total CPU time required to execute the program may increase. This is because the total CPU time is at least as large as the CPU time when the program is executed on a single processor, and the overhead time for multi-processing may increase the total CPU time.

## Impediments to Improvements

Speed improvements from multi-processing using LF95 PRO come from splitting up loops among the available processors. Impediments to performance improvements include the following:

- Overhead for initiating and managing threads on secondary processors.
- Lack of large arrays and loops operating on them.
- I/O intensive rather than computationally intensive programs.
- Potential for incorrect results.
- Other unparallelizable loops.

These impediments are discussed in the sections below.

### Overhead

Time is spent whenever your program starts up or shuts down a thread (a separate stream of execution) on a secondary processor. This time can outweigh the time gained by running part of the code on a secondary processor if the work to be done on that processor is not significant.

### Lack of Large Arrays

If your program does not spend the bulk of its time in computationally intensive loops then there is not adequate work to divide among the processors. Your program will likely run at least as fast without parallelization. For example, if half of your program's time is spent in parallelizable loops then the best time savings you can expect by parallelization on two processors is 25%. If your program takes two minutes to run serially, and half of its time is spent in parallelizable loops, then the theoretically optimal parallel run time is one minute and thirty seconds.

### I/O Intensive Programs

If your program spends much of its time reading or writing files or waiting for user input then any speed increase due to parallelization will likely be dwarfed by the time spent doing I/O. Your program will likely not show a significant performance improvement.

### Potential for Incorrect Results

Certain loops can be analyzed sufficiently to be parallelized by the compiler without input from the programmer. However, many loops have data dependencies that would prevent automatic parallelization because of the potential for incorrect results. For that reason, LF95 PRO includes optimization control lines (see *"Optimization Control Line"* on page 77) and OpenMP directives (see *"OpenMP"* on page 89), with which the programmer can provide the information necessary for the compiler to parallelize otherwise unparallelizable loops.

### Other Unparallelizable Loops

Some loops cannot be parallelized for other reasons discussed later in this chapter. Sometimes recoding a loop to move a statement or group of statements outside the loop will allow that loop to be parallelized.

## Hardware for Multi-Processing

A computer environment with two CPUs that operate independently and simultaneously is necessary to save elapsed time by multi-processing. A multi-processing program can be executed on hardware with only a single CPU; however, the elapsed time will not be less than the execution time for a comparable program written without multi-processing features.

# Automatic Parallelization

With automatic parallelization, DO loops and array operations are parallelized without the programmer making any modifications to the program. This makes it easy to migrate source programs to other processing systems as long as the program conforms with the Fortran standard.

## Compiler Options for Automatic Parallelization

There are four compiler options for automatic parallelization. They are --parallel, --threads, --threadstack, and --ocl. These options are documented in *"Compiler and Linker Options"* on page 14.

## Environment Variables

The following section details the various environment variables that can be set to alter the way a parallel program executes.

### Environment Variable PARALLEL

When the environment variable PARALLEL is set, its value must be less than or equal to the number of CPUs active at run-time. (It is called the number of active CPUs.)

**Note:**

If --threads is specified during compilation, the value of PARALLEL must be equal to the argument to --threads and the number of active CPUs must be greater than or equal to the argument to --threads. If the environment variable PARALLEL is not set, the argument to --threads. must be the same as the number of active CPUs.

### Environment Variable THREAD_STACK_SIZE

When the environment variable THREAD_STACK_SIZE is set, it sets the stack size in Kilobytes for each thread stack. Local variables in DO loops and array operations are allocated on the stack. You may need to extend the stack size if there are many of these local variables. The default stack size for each thread is the same as that of the executable. The compiler option --threadstack and environment variable THREAD_STACK_SIZE can change the stack size of each thread. The compiler option --threadstack takes precedence over the environment variable THREAD_STACK_SIZE.

### Examples of Compilation and Execution

```
% lf95 --info --parallel --ocl test1.f
% a.out
```

In example above, automatic parallelization and optimization control lines (OCLs) are in effect during compilation. This program is executed using all active CPUs on the machine.

```
% lf95 --parallel test2.f
5001-i: "test2.f", line 2: DO loop with index i parallelized.
% setenv PARALLEL 2
% a.out
% setenv PARALLEL 4
% a.out
```

In this second example, the environment variable PARALLEL is set to 2 and the program executes with two CPUs. Next, the environment variable PARALLEL is set to 4 and the program executes with four CPUs.

## Details of Multi-Processing

This section describes multi-processing in more detail.

### Targets for Automatic Parallelization

Target statements of the automatic parallelization are DO loops (including nested DO loops) and array operations (array expressions and array assignments).

### Loop Slicing

Automatic parallelization may slice a DO loop into several pieces. The elapsed execution time is reduced by executing the sliced DO loops in parallel.

```
do i = 1, 50000
  a(i) = b(i) + c(i)
end do
```

Different iterations of the DO loop can be executed on different CPUs at the same time.

CPU 1:

```
do i1 = 1, 25000
  a(i1) = b(i1) + c(i1)
end do
```

CPU 2:

```
do i2 = 25001, 50000
  a(i2) = b(i2) + c(i2)
end do
```

### Array Operations and Automatic Parallelization

Automatic parallelization also targets statements with array operations (array expressions and array assignments).

```
integer a(1000), b(1000)
a = a + b
```

Half of the operations are made on one CPU and half are made on the other.

CPU 1:

```
a(1:500) = a(1:500) + b(1:500)
```

CPU 2:

```
a(501:1000) = a(501:1000) + b(501:1000)
```

### Automatic Loop Slicing by the Compiler

LF95 parallelizes a DO loop if the order of data references will be the same as with serial execution. LF95 assures that the result of a multi-processing program is the same as if the program were processed serially.

The next example is a DO loop that is not amenable to loop slicing. In this DO loop, when the DO variable I is 5001, it is necessary to have the value of array element A(5000).

```
do i = 2,10000
  a(i) = a(i-1) + b(i)
end do
```

The following loop slicing cannot happen with the code above:

CPU 1:

```
do i = 2,5000
  a(i) = a(i-1) + b(i)
end do
```

CPU 2:

```
do i = 5001, 10000
  a(i) = a(i-1) + b(i)
end do
```

A(5000) is not available to CPU2 and the loop will not be sliced

### Loop Interchange and Automatic Loop Slicing

When a nested DO loop is sliced, LF95 attempts to parallelize the outermost loop if it can. LF95 selects a DO loop that can be sliced and interchanges it with the outermost possible loop. The purpose of this is to reduce the overhead of multi-processing and improve execution performance.

The next figure shows an example of loop interchange for a nested loop. It is possible to slice the inner loop with control variable J. The frequency of multi-processing control can be reduced by interchanging it with the outer loop.

```
do i = 2, 10000
  do j = 1, 10
    a(i,j) = a(i-1,j) + b(i,j)
  end do
end do
```

With loops interchanged, this becomes:

```
do j = 1, 10
  do i = 2, 10000
    a(i,j) = a(i-1,j) + b(i,j)
  end do
end do
```

When parallelized, this becomes:

CPU 1:

```
do j = 1, 5
  do i = 2, 10000
    a(i,j) = a(i-1,j) + b(i,j)
  end do
end do
```

CPU 2:

```
do j = 6, 10
  do i = 2, 10000
    a(i,j) = a(i-1,j) + b(i,j)
  end do
end do
```

### Loop Distribution and Automatic Loop Slicing

In the next example, the references to array A cannot be sliced, because the order of data references would be different from the data reference order in serial execution. Array B can be sliced, because the order of data references is the same as for serial execution. For this case, the statement where array A is defined and the statement where array B is defined are separated into two DO loops, and the DO loop where array B is defined is parallelized.

```
do i = 1, 10000
  a(i) = a(i-1) + c(i)
  b(i) = b(i) + c(i)
end do
```

With the loop distributed this becomes:

```
do i = 1, 10000
  a(i) = a(i-1) + c(i)
end do

do i = 1, 10000
  b(i) = b(i) + c(i)
end do
```

The second loop is then parallelized:

CPU 1:

```
do i = 1, 5000
  b(i) = b(i) + c(i)
end do
```

CPU 2:

```
do i = 5001, 10000
  b(i) = b(i) + c(i)
end do
```

### Loop Fusion and Automatic Loop Slicing

In the next example, there are DO loops in sequence having the same DO loop control. In this case, the overhead of the DO loop control and the frequency of multi-processing control can be reduced by merging those two loops into a single loop.

```
do i = 1, 10000
  a(i) = b(i) + c(i)
end do

do i = 1, 10000
  d(i) = e(i) + f(i)
end do
```

With loops fused this becomes:

```
do i = 1, 10000
  a(i) = b(i) + c(i)
  d(i) = e(i) + f(i)
end do
```

When parallelized, this becomes:

CPU 1:

```
do i = 1, 5000
  a(i) = b(i) + c(i)
  d(i) = e(i) + f(i)
end do
```

CPU 2:

```
do i = 5001, 10000
  a(i) = b(i) + c(i)
  d(i) = e(i) + f(i)
end do
```

### Loop Reduction

Loop reduction slices the DO loop, changing order of the operations (addition and multiplication, etc.). Note that loop reduction may cause small differences in execution results.

Loop reduction optimization is applied if there is one of the following operations in the DO loop:

- SUM: `S=S+A(I)`
- PRODUCT: `P=P*A(I)`
- DOT PRODUCT: `P=P+A(I)*B(I)`
- MIN: `X=MIN(X,A(I))`
- MAX: `Y=MAX(Y,A(I))`
- OR: `N=N.OR. A(I)`
- AND: `M=M.AND.A(I)`

The next example shows loop reduction and automatic loop slicing.

```
sum = 0
do i = 1, 10000
  sum = sum + a(i)
end do
```

Parallelized becomes:

CPU 1:

```
sum1 = 0
do i = 1, 5000
  sum1 = sum1 + a(i)
end do
```

CPU 2:

```
sum2 = 0
do i = 5001, 10000
  sum2 = sum2 + a(i)
end do
```

The partial sums are added:

```
sum = sum + sum1 + sum2
```

#### Restrictions on Loop Slicing

The following types of DO loop are not targets for loop slicing.

1. Loops where it is forecast that the elapsed time would not be reduced.
2. The loop contains operations of a type not suitable for loop slicing.
3. The loop contains a procedure reference.
4. The loop is too complicated.
5. The loop contains an I/O statement.
6. Loops where the order of data references would be different from that of serial execution.

#### Debugging

Multi-threaded programs cannot be debugged using fdb.

## Optimization Control Line

LF95 PRO offers an optimization control line (OCL) feature that helps automatic parallelization. The optimization control line is used by the programmer to identify constructs that may be executed in parallel. Because OCLs are in Fortran comments, programs with OCLs can still be standard-conforming and can be compiled with other compilers that do not support OCLs.

The optimization control lines (OCLs) take effect when both --parallel and --ocl options are specified.

### Optimization Control Specifier

The optimization control lines (OCLs) have several functions depending on the optimization control specifier.

### Syntax of OCL

Columns 1-5 of an optimization control line (OCL) must be "!OCL ". One or more optimization control specifiers follow.

!OCL *i* [,*i*] ....

where each *i* is an optimization control specifier, either SERIAL, PARALLEL, DISJOINT, TEMP, or INDEPENDENT (see *"Optimization Control Specifier"* on page 78).

### Position of OCL

The position of the OCL depends on the optimization control specifier.

The OCL for automatic parallelization must occur at a total-position or loop-position. Total-position and loop-position are defined as follows:

- Total-position: the top of each program unit.
- Loop-position: immediately before a DO loop. However, more than one OCL may be specified at loop-position and comment lines may be specified between the OCLs and the DO loop.

```
!ocl serial  <------------------ total-position
      subroutine sub(b, c, n)
      integer a(n), b(n), c(n)
      do i = 1, n
        a(i) = b(i) + c(i)
      end do
      print*, fun(a)
!ocl parallel  <---------------- loop-position
      do i = 1, n
        a(i) = b(i) * c(i)
      end do
      print*, fun(a)
      end
```

### Automatic Parallelization and Optimization Control Specifiers

An optimization control specifier becomes ineffective for a DO loop that is not a target of loop slicing, even if the optimization control specifier for automatic parallelization is specified.

### Optimization Control Specifiers

The following optimization control specifiers are used to enhance automatic parallelization:

- SERIAL

- PARALLEL
- DISJOINT
- TEMP
- INDEPENDENT

### SERIAL

The SERIAL specifier is used to inhibit DO loop slicing.

For instance, if the programmer knows that serial execution of a DO loop is faster than parallel execution, perhaps because the iteration count will always be small, the programmer may specify the SERIAL specifier for the DO loop.

**Syntax:**

!OCL SERIAL

The SERIAL specifier may be specified at the loop position or the total position.  The effect of SERIAL depends on its position.

- At the loop position

SERIAL inhibits loop slicing for the DO loop (including any nested loops) corresponding to the OCL.

- At the total position

SERIAL inhibits loop slicing for all loops in the program unit containing the OCL.

In the following program, if loop 2 should not be sliced, loop slicing can be disabled by specifying SERIAL.

the letter p on the left side of the source program marks the parallelized statements.

```
p      do j = 1, 10
p        do i = 1, l   ! <----------- loop 1
p          a1(i,j) = a1(i,j) + b1(i,j)
p          c1(i,j) = c1(i,j) + d1(i,j)
p          e1(i,j) = e1(i,j) + f1(i,j)
p          g1(i,j) = g1(i,j) + h1(i,j)
p        end do
p      end do

p      do j=1, 10
p        do i=1, m   ! <------------ loop 2
p          a2(i,j) = a2(i,j) + b2(i,j)
p          c2(i,j) = c2(i,j) + d2(i,j)
p          e2(i,j) = e2(i,j) + f2(i,j)
p          g2(i,j) = g2(i,j) + h2(i,j)
p        end do
p      end do
```

```
p      do j=1, 10
p        do i=1, n   ! <------------ loop 3
p          a3(i,j) = a3(i,j) + b3(i,j)
p          c3(i,j) = c3(i,j) + d3(i,j)
p          e3(i,j) = e3(i,j) + f3(i,j)
p          g3(i,j) = g3(i,j) + h3(i,j)
p        end do
p      end do

p      do j = 1, 10
p        do i = 1, l   ! <------------ loop 1
p          a1(i,j) = a1(i,j) + b1(i,j)
p          c1(i,j) = c1(i,j) + d1(i,j)
p          e1(i,j) = e1(i,j) + f1(i,j)
p          g1(i,j) = g1(i,j) + h1(i,j)
p        end do
p      end do

 !ocl serial
       do j = 1, 10
         do i = 1, m   ! <------------ loop 2
           a2(i,j) = a2(i,j) + b2(i,j)
           c2(i,j) = c2(i,j) + d2(i,j)
           e2(i,j) = e2(i,j) + f2(i,j)
           g2(i,j) = g2(i,j) + h2(i,j)
         end do
       end do

p      do j = 1, 10
p        do i = 1, n   <-------------- loop 3
p          a3(i,j) = a3(i,j) + b3(i,j)
p          c3(i,j) = c3(i,j) + d3(i,j)
p          e3(i,j) = e3(i,j) + f3(i,j)
p          g3(i,j) = g3(i,j) + h3(i,j)
p        end do
p      end do
```

### PARALLEL

The PARALLEL specifier is used to reverse the effect of the SERIAL specifier and enables loop slicing.

**Syntax:**

```
!OCL PARALLEL
```

The PARALLEL specifier can be placed at the loop position or the total position.

The effect of PARALLEL depends on its position.

- At the loop position

PARALLEL allows loop slicing for the DO loop (and any nested loops) corresponding to the OCL.

- At the total position

PARALLEL allows loop slicing for all loops in the program containing the OCL.

In the following example, if only loop 2 should be sliced, it can be sliced by specifying PARALLEL together with SERIAL as shown.

The letter P on the left side of the source program marks the parallelized statements.

```
!ocl serial  <------------ total position
            .
            .
            .
        do j = 1, 10
          do i = 1, l   ! <----------- loop 1
            a1(i,j) = a1(i,j) + b1(i,j)
            c1(i,j) = c1(i,j) + d1(i,j)
            e1(i,j) = e1(i,j) + f1(i,j)
            g1(i,j) = g1(i,j) + h1(i,j)
          end do
        end do
!ocl parallel
 p      do j = 1, 10
 p        do i = 1, m   ! <----------- loop 2
 p          a2(i,j) = a2(i,j) + b2(i,j)
 p          c2(i,j) = c2(i,j) + d2(i,j)
 p          e2(i,j) = e2(i,j) + f2(i,j)
 p          g2(i,j) = g2(i,j) + h2(i,j)
 p        end do
 p      end do
       do j = 1, 10
          do i = 1, n   ! <----------- loop 3
            a3(i,j) = a3(i,j) + b3(i,j)
            c3(i,j) = c3(i,j) + d3(i,j)
            e3(i,j) = e3(i,j) + f3(i,j)
            g3(i,j) = g3(i,j) + h3(i,j)
          end do
        end do
```

### DISJOINT

The DISJOINT specifier indicates that the order of data references (references to arrays in the DO loop) is the same whether executed serially or in parallel.

As a result, it is possible to slice a DO loop that would not be sliced otherwise because the compiler would be unable to determine the order of data references.

**Syntax:**

!OCL DISJOINT [ (*a* [,*a*]...) ]

Here, "*a*" is the array name for which loop slicing is possible. A wild-card specification is usable in "*a*". If the array name is omitted, DISJOINT becomes effective for all arrays within the range of the DO loop. See *"Wild Card Specification"* on page 85 for the wild-card syntax.

The DISJOINT specifier can be placed at the loop position or the total position.

The effect of DISJOINT depends on its position.

- At the loop position

DISJOINT promotes loop slicing for the DO loop (and all nested loops) corresponding to the OCL.

- At the total position

DISJOINT promotes loop slicing for all loops in the program unit. Consider the following code:

```
do j = 1, 1000
  do i = 1, 1000
    a(i,l(j)) = a(i,l(j)) + b(i,j)
  end do
end do
```

Because the subscript expression of array A is another array element L(J), the system cannot determine whether there is a problem if array A is sliced. Therefore, this system does not slice the outer DO loop.

If the programmer knows that there is no problem if array A is sliced, the outer DO loop will be sliced if DISJOINT is used as shown in the example below.

The letter P shown on the left side of the source program marks the parallelized statements.

```
!ocl disjoint(a)
 p      do j = 1, 1000
 p        do i = 1, 1000
 p          a(i,l(j)) = a(i,l(j)) + b(i,j)
 p        end do
 p      end do
```

**Note:**
If an array which cannot be sliced is marked DISJOINT by mistake, LF95 may perform an incorrect loop slicing and the program results may be incorrect.

### TEMP

The TEMP specifier is used to indicate to the system that the variables listed are used temporarily in the DO loop.

As a result, the execution performance of the parallelized DO loop can be improved.

**Syntax:**

!OCL TEMP [ (*s* [,*s*]...) ]

Here, "*s*" is a variable name used temporarily in a DO loop. A wild card specification is usable in "*s*". If the variable name is omitted, TEMP becomes effective for all scalar variables within the range of the DO loop. See *"Wild Card Specification"* on page 85 for the wild-card syntax.

The TEMP specifier can be placed at the loop position or the total position.

The effect of TEMP depends on its position.

- At the loop position

TEMP indicates that the variables in the DO loop corresponding to the OCL are temporary variables.

- At the total position

TEMP indicates that the variables of all loops in the program unit containing the OCL are temporary variables.

In the example below, because T is a common variable, LF95 must assume that variable T is referenced in subroutine SUB even if T is used only in the DO loop. LF95 adds code to guarantee that T has the correct value at the end of the parallelized DO loop.

The letter P shown on the left side of the source program marks the parallelized statements.

```
          common t
                .
                .
                .
p         do j = 1, 50
p           do i = 1, 1000
p             t = a(i,j) + b(i,j)
p             c(i,j) = t + d(i,j)
p           end do
p         end do
                .
                .
                .
          call sub
```

If the programmer knows that the value of T at the end of the DO loop is not needed in subroutine SUB, the programmer may specify the TEMP specifier with T as shown in the following code. As a result, the execution performance improves, because the instruction which corrects the value of T becomes unnecessary at the end of the DO loop.

```
              common t
                    .
                    .
                    .
      !ocl temp(t)
       p      do j = 1, 50
       p        do i = 1, 1000
       p          t = a(i,j) + b(i,j)
       p          c(i,j) = t + d(i,j)
       p        end do
       p      end do
                    .
                    .
                    .
              call sub
```

**Note:**
If a variable that is not used temporarily is described in a TEMP specifier by mistake, LF95 may do an incorrect loop slicing and the program results may be incorrect.

### INDEPENDENT

The INDEPENDENT specifier is used to indicate to LF95 that parallel execution is the same as serial execution even if a procedure is called in the DO loop. As a result, the DO loop that contains the procedure is suitable for loop slicing.

**Syntax:**

!OCL INDEPENDENT [ (*e* [,*e*]...) ]

Here, "*e*" is a procedure name which does not inhibit loop slicing. The wild card specification is usable in "*e*". If the procedure name is omitted, INDEPENDENT becomes effective for all procedures within the range of the DO loop. See *"Wild Card Specification"* on page 85 for wild card specification.

Note that the procedure *e* must be compiled with the --parallel option.

The INDEPENDENT specifier can be placed at the loop position or the total position.

The effect of INDEPENDENT depends on its position.

- At the loop position

INDEPENDENT allows loop slicing for the DO loop (and all nested loops) corresponding to the OCL.

- At the total position

INDEPENDENT allows loop slicing for all loops in the program containing the OCL. Consider the following code:

```
do i = 1, 10000
  j = i
  a(i) = fun(j)
end do
   .
   .
end
function fun(j)
fun = sqrt(real(j**2+3*j+6))
end
```

In the program above, because the procedure "FUN" is called in the DO loop, the system cannot determine whether the DO loop can be parallelized.

If the programmer knows that there is no problem even if the DO loop which contains the reference to the procedure "FUN" is sliced, the DO loop can be sliced by using INDEPENDENT as shown in the code below.

The letter P shown on the left side of the source program marks the parallelized statements.

```
  !ocl independent(fun)
 p      do i = 1,1000
 p        j = i
 p        a(i) = fun(j)
 p      end do
           .
           .
        end
        function fun(j)
        fun = sqrt(real(j**2+3*j+6))
        end
```

**Note:**
If a procedure that cannot be sliced is described in an INDEPENDENT specifier by mistake, LF95 may perform an incorrect loop slicing and program results may be incorrect.

### Wild Card Specification

In the operand of the following optimization control specifiers, a wild card may be specified for a variable name or a procedure name:

- DISJOINT
- TEMP
- INDEPENDENT

The wild card specification is a combination of the special wild card characters and alphanumeric characters. The effect is the same as specifying all of the variable names and procedure names that agree with the wild card expression. There are two wild card characters, "*" and "?", and they match the following character strings.

- "*" matches any character string of one or more alphanumeric characters.

- "?" matches any single alphanumeric character.

A wild card specification cannot contain more than one wild card character.

```
!ocl temp(w*)
```

In this example, `w*` matches any variable beginning with w and having a length of two or more characters. For example, the variable names `work1`, `w2`, and `work3` are included in this specification.

```
!ocl disjoint(a?)
```

In this example, `a?` matches any two-character array name which has `a` for the first character. For example, the array names `a1`, `a2`, and `aa` are included in this specification. The array name `abc` is not included in this specification because its length is not two.

```
!ocl independent(sub?)
```

In this example, `sub?` matches any four-character procedure name whose first three character are `sub`. For example, procedure names sub1, `sub2`, and `sub9` are included in this specification.

## Notes on Parallelization

This section explains some specifics about the parallelization facility.

### --threads

When the number of CPUs executed in parallel is specified by the --threads compiler option, the argument to --threads must have the same value as the value of the PARALLEL environment variable. If the PARALLEL environment variable is not set, the value of the argument to --threads must be the same value as the number of CPUs active at run-time.

The example below shows an invalid use of the --threads compiler option when the number of active CPUs is four. If an invalid value for --threads is specified, execution results may be incorrect.

In the following incorrect example, the value of *N* and the value of PARALLEL are different.

```
% setenv PARALLEL 2
% lf95 --parallel --threads 4 a.f
```

In the following example, execution results may be incorrect if the number of active CPUs is not equal to two.

```
% lf95 --parallel --threads 2 a.f
```

### Multi-Processing of Nested DO Loops

If there is a parallelized DO loop in a procedure that is called from within another parallelized DO loop, a nest of parallelized DO loops is generated. A program that contains such DO loops must not be compiled with the --threads compiler option.

The following is an example in which the parallelized DO loop should be executed serially. If a source program that contains such DO loops is compiled with the --threads compiler option, the result may be incorrect.

```
file: a.f
!ocl independent(sub)
       do i = 1,100    ! <------ executed in parallel
         j = i
         call sub(j)
       end do
          :
       end
       subroutine sub(n)
          :
       do i = 1, 10000 ! < ----- should be executed serially
         a(i) = 1 / b(i)**n
       end do
          :
       end
```

The result may be incorrect if the source program `a.f` is compiled as follows.

```
% lf95 --parallel --threads 4 a.f       (invalid use)
```

To prevent such a mistake, specify the optimization control line `!OCL SERIAL` in the procedure that is called from within the parallelized DO loop.

```
!ocl serial
      subroutine sub(n)
         :
      do i = 1,10000
        a(i) = 1 / b(i)**n
      end do
         :
      end
```

**Loop Reduction Effects**

When --parallel is specified as a compiler option, the result of execution may be different from the result of serial execution. The reason for this is that as a result of loop reduction, the operation order may be different between the parallel execution and the serial execution.

The following illustrates the loop reduction optimization.

```
sum = 0
do i = 1, 10000
  sum = sum + a(i)
end do
```

When parallelized, this becomes:

CPU 1:

```
      sum1 = 0
      do i = 1, 5000
        sum1 = sum1 + a(i)
      end do
```

CPU 2:

```
      sum2 = 0
      do i = 5001, 10000
        sum2 = sum2 + a(i)
      end do
```

Then the partial sums are added:

```
sum = sum + sum1 + sum2
```

The variable SUM accumulates the values A(1) to A(10000) in order with serial execution. In parallel execution, SUM1 accumulates the values A(1) to A(5000), and SUM2 accumulates the values A(5001) to A(10000) at the same time. After that, the sum of SUM1 and SUM2 is added to SUM.

Loop reduction optimization may cause a side effect (a different result due to rounding) in the execution result, because the order of adding the array elements is different between parallel execution and serial execution.

### Invalid Usage of Optimization Control Line

The following program specifies DISJOINT by mistake for array A. The execution result will be incorrect when array A is sliced, because the order of the data references for array A is different from the order of data references for serial execution.

```
!ocl disjoint(a)
    do i = 2,10000
      a(i) = a(i-1) + b(i)
    end do
```

The following program specifies `TEMP` by mistake for variable `T`. The correct value will not be assigned to variable `last`, because LF95 does not guarantee a correct value of variable `T` at the end of the DO loop.

```
!ocl temp(t)
    do i = 1, 1000
      t = a(i) + b(i)
      c(i) = t + d(i)
    end do
    last = t
```

The following program specifies `INDEPENDENT` by mistake for procedure `SUB`. The execution result may be incorrect when array A is sliced, because the order of the data references for array A is different from the data references for serial execution.

```
    common a(1000), b(1000)
!ocl independent(sub)
    do i = 2, 1000
      a(i) = b(i) + 1.0
      call sub(i-1)
    end do
     ...
    end
    subroutine sub(j)
    common a(1000)
    a(j) = a(j) + 1.0
    end
```

**Multi-processing I/O Statements and Intrinsic Procedure References**

If there is an I/O statement, an intrinsic subroutine or function reference that is not suitable for loop slicing in a procedure that is called in a parallelized DO loop, execution of the program will produce incorrect results. The execution performance of the multi-processing program may decrease due to the overhead of parallel execution. Also, the result of the I/O statement may be different from the result of serial execution.

The following is an example in which an I/O statement occurs in a procedure that is called in a parallelized DO loop.

```
file: a.f
 !ocl independent(sub)
   do i = 1, 100
     j = i
     call sub(j)
   end do
      :
   end
   recursive subroutine sub(n)
      :
   print*, n
      :
   end
```

## OpenMP

The compiler supports OpenMP v.2.0 directives. This section describes parallelization using OpenMP. Refer to the OpenMP Fortran specification included with LF95 in PDF format for non-implementation-specific information on OpenMP. The following website includes comprehensive information on OpenMP:

```
http://www.openmp.org/
```

It is assumed that the reader has an understanding of OpenMP. LF95's implementation of OpenMP is described below.

## Compilation

There are three compiler options for OpenMP parallelization. They are --openmp, --threadstack, and --threadheap. These options are documented in *"Compiler and Linker Options"* on page 14.

## Environment Variables

OpenMP specifies a number of environment variables, which are described in the OpenMP documentation at `http://www.openmp.org`. Along with the OpenMP environment variables, this implementation has:

**FLIB_FASTOMP={ true | false }**

If the environment variable FLIB_FASTOMP is present and set to true or 1, the compiler will link with high-speed runtime libraries optimized for OPENMP..

**FLIB_SPINWAIT=*wait_time***

The user can specify the mode of waiting threads using the environment variable FLIB_SPINWAIT.

*wait_time* denotes how long to wait before suspending the thread, and is specified in seconds by appending the letter "s" to *wait_time*, or is specified in milliseconds by appending the letters "ms" to *wait_time*. *wait_time* may also have the value `unlimited`, which is the default value. If the value of *wait_time* is `unlimited`, the waiting thread is never suspended. If the value of *wait_time* is `0`, the waiting thread is immediatelty suspended. Use of a large or `unlimited` *wait_time* will result in a faster elapsed time for program execution, but will cause the total CPU time consumed to increase.

**THREAD_STACK_SIZE=*num***

The user can specify the size of stack for each thread using the environment variable THREAD_STACK_SIZE.

*num* is a number in the range 16 ≤ *num* ≤ 2048**.**

The --threadstack compiler option overrides this environment variable.

## Implementation Specifications

This section gives details on features that are left processor-dependent by the OpenMP specification along with other specifications and restrictions.

### Nesting of Parallel Regions

Nesting of parallel regions is supported.

### Dynamic Thread Adjustment Features

Dynamic thread adjustment features are supported, and are on by default.

### Number of Threads

The number of threads for OpenMP is decided with the following priority.

1. OMP_SET_NUM_THREADS service routine
2. Environment variable OMP_NUM_THREADS
3. Environment variable PARALLEL
4. One thread

### SCHEDULE Clause

If the SCHEDULE Clause is omitted, the default is SCHEDULE(STATIC).

### OMP_SCHEDULE Environment Variable

When the OMP_SCHEDULE environment variable is omitted, a DO directive or PARALLEL DO directive having the schedule type RUNTIME will default to SCHEDULE(STATIC).

### ASSIGN and Assigned GO TO Statements

An ASSIGN statement within an OpenMP block cannot refer to a statement label that is outside of the OpenMP directive block. Also, a statement label in an OpenMP directive block cannot be referred to by an ASSIGN statement that is outside of the OpenMP directive block.

Jumping into or out of a directive block area using an assigned GO TO statement is not supported.

### Additional Functions and Operators in ATOMIC directive and REDUCTION Clause

The following intrinsic functions and operators can be specified in an ATOMIC directive or REDUCTION clause.

|  |  |
|---|---|
| Intrinsic functions | : AND, OR |
| Operators | : .XOR., .EOR. |

### FORALL construct

In a FORALL construct, OpenMP directives cannot be used.

### THREADPRIVATE

When using the THREADPRIVATE directive, a given common block must be defined the same in all program units. A common block specified as THREADPRIVATE cannot have its size extended.

### IF Clause for PARALLEL Directive

When the IF clause for a PARALLEL directive is not true, the PARALLEL directive is ignored. Therefore, no team of threads is created. However, the PARALLEL directive remains in effect.

### Inline Expansion

The following procedures are not inline expanded.

- User-defined procedures that include OpenMP Fortran directives.
- User-defined procedure that are referred to in OpenMP directives.

### Internal Procedure Calling from Parallel Region

A variable in the host procedure referenced in an internal procedure that is called in a parallel region is regarded as SHARED even if it is privatized in the parallel region.

```
:
i = 1        ! this i is shared
!$omp parallel private(i)
i = 2        ! i is private
print*, i    ! i is private
call proc    ! i is private
!$omp end parallel
contains
  subroutine proc()
  :          ! i is shared
  print*, i  ! i is shared
  :          ! i is shared
  end subroutine
:
```

### DO Variable for Serial DO Loop in Parallel Region

When the DO variable of a serial DO loop within a parallel region is marked as "SHARED", it is privatized in the scope of the DO loop.

```
!$omp parallel shared(i)
i = 1             ! i is shared
do i = 1, n       ! i is private
  :               ! i is private
end do            ! i is private
print*, i         ! i is shared
!$omp end parallel
```

```
!$omp parallel private(i)
i = 1             ! i is private
do i = 1, n       ! i is private
  :               ! i is private
end do            ! i is private
print*, i         ! i is private
!$omp end parallel
```

### Statement Function Statement

A variable that appears in a statement function statement cannot have the PRIVATE, FIRSTPRIVATE, LASTPRIVATE, REDUCTION, or THREADPRIVATE attribute.

### Namelist Group Object

A variable declared as a namelist group object cannot have the PRIVATE, FIRSTPRIVATE, LASTPRIVATE, REDUCTION, or THREADPRIVATE attribute.

### Materialization of Parallel Region

Internal procedures are SCHEDULE(STATIC).

The generated internal procedure has the name "_*n*_", where *n* is a consecutive number.

### Automatic Parallelization with OpenMP

The --openmp option and the --parallel option may be specified at the same time. The --parallel option is ignored in any program unit that contains OpenMP directives.

### Debugging

Multi-threaded programs cannot be debugged using fdb.

# 6 Automake (PRO version only)

## Introduction

### What Does It Do?

AUTOMAKE is a simple-to-use tool for re-building a program after you have made changes to the Fortran and/or C source code. It examines the creation times of all the source, object and module files, and recompiles wherever it finds that an object or module file is non-existent, empty or out of date. In doing this, it takes account not only of changes or additions to the source code files, but also changes or additions to MODULEs and INCLUDEd files - even when nested. For example, if you change a file which is INCLUDEd in half a dozen source files, AUTOMAKE ensures that these files are re-compiled. In the case of Fortran 95, AUTOMAKE ensures that modules are recompiled from the bottom up, taking full account of module dependencies.

### How Does It Do That?

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

### How Do I Set It Up?

The operation of AUTOMAKE is controlled by a configuration file which contains the default compiler name and options, INCLUDE file search rule, etc. For simple situations, where the source code to be compiled is in a single directory, and builds into a single executable, it will probably be possible to use the system default configuration file. In that case there is no need for any customization of AUTOMAKE— just type `am` to update both your program and the dependency file.

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

### What Can Go Wrong?

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

## Running AUTOMAKE

To run AUTOMAKE, simply type `am`. If there is a configuration file (`AUTOMAKE.FIG`) in the current directory, AUTOMAKE reads it.

## The AUTOMAKE Configuration File

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

*keyword=value*

or

*keyword*

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

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

**LF95**

Equivalent to specifying the default LF95 compile and link commands.

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

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

**FILES=**

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

```
FILES=*.f90
```

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

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

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

Note also that, if multiple FILES= lines are specified, then the %RF place marker (place markers will be explained in the next section) cannot be used in any COMPILE= lines.

**COMPILE=**

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

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

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

**Table 10: COMPILE= Place Markers**

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

```
COMPILE=lf95 -c %fi --mod %mo
COMPILE=lf95 -c @%rf --include %id
```

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

**TARGET=**

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

```
TARGET=/execs/MYPROG.EXE
```

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

**LINK=**

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

```
LINK=lf95 %ob -o %ex --mod %mo'

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

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

**Table 11: LINK= Place Markers**

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

**INCLUDE=**

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

```
INCLUDE=/include:/include/sys
```

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

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

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

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

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

```
OBJDIR=OBJ/
```

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

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

```
OBJEXT=abc
```

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

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

```
MODULE=MODS/
```

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

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

```
DEP=thisprog.dep
```

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

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

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

**MAKEMAKE**
Specifies that AUTOMAKE should create a text file called `automake.mak` containing dependency information.

**DEBUG**
Causes AUTOMAKE to write debugging information to a file called `automake.dbg`.

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

**CHECK=**
May be used to specify a command to be inserted after each compilation. A typical application would be to check for compilation errors.

## Multi-Phase Compilation

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

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

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

```
# Example Configuration file for Multi-Phase
# Compilation
# Compilation 1 - files in current directory
LF95
INCLUDE=/include
FILES=*.f90
OBJDIR=obj
COMPILE=lf95 -c %fi -I %id -o %od%sf%oe --tp -O1
AND
# Compilation 2 - files in utils/
# INCLUDE= defaults to previous value (/include)
# if OBJDIR= were not set, it would default to utils (NOT obj)
FILES=utils/*.f90
OBJDIR=utils/obj
COMPILE=lf95 -c %fi -I %id -o %od%sf%oe --sav --chk
# Relink
TARGET=a.out
LINK=lf95 %ob -o %ex
```

## Automake Notes

- As AUTOMAKE executes, it issues brief messages to explain the reasons for all compilations. It also indicates when it is scanning through a file to look for INCLUDE statements.
- If for any reason the dependency file is deleted, AUTOMAKE will create a new one. Execution of the first AUTOMAKE will be slower than usual, because of the need to regenerate the dependency data.
- AUTOMAKE recognizes the INCLUDE statements in all common variants of Fortran and C, and can be used with both languages.
- When AUTOMAKE scans source code to see if it contains INCLUDE statements, it recognizes the following generalized format:
- Optional spaces at the beginning of the line followed by an optional compiler control character, '%', '$' or '#', followed by the word INCLUDE (case insensitive) followed by an optional colon followed by the file name, optionally enclosed between apostrophes, quotes or angled brackets. If the file name is enclosed in angled brackets, it is assumed to be in one of the directories specified using the SYSINCLUDE keyword. Otherwise, AUTOMAKE looks in the source file directory, and if it is not there, in the directories specified using the INCLUDE keyword.
- If AUTOMAKE cannot find an INCLUDE file, it reports the fact to the screen and ignores the dependency relationship.

- AUTOMAKE is invoked using a script file called `am`. There is seldom any reason to modify the script file, though it is very simple to do so if required. It consists of two (or three) operations:

1. Execute AUTOMAKE. AUTOMAKE determines what needs to be done in order to update your project and writes a script file to do it. The options which may be appended to the AUTOMAKE command are:

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

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

2. Execute the command file (`automake.tmp`) created by AUTOMAKE.
3. Delete the command file created by AUTOMAKE. This step is, of course, optional.

# 7 Utility Programs

This chapter documents the following utility programs:

- `fot`
- `hdrstrip.f90`
- `sequnf.f90`
- `tryblk.f90`

## fot

**Usage:**

`fot` *[file1] [file2]*

`fot` is a program that is used for converting files created by LF95, opened as CARRIAGE-CONTROL='FORTRAN', into a form suitable for printing. `fot` interprets the first character of each line of *file1* as a Fortran carriage control character to be used for printing, producing a file *file2* in Linux format. The first character of each line of *file1* causes the following modifications:

**blank:** The blank is deleted, which causes the line to be printed with single spacing. A line of all blanks is converted to a line with no characters.

**0:** The character is changed to a new-line character, which causes the line to be printed with double spacing.

**1:** The character is changed to the new-page character, which causes the line to be printed at the beginning of a new page.

**+:** If it is the first line of a file, the character is deleted. Otherwise, the character is replaced by a carriage-return character, which causes the line to be printed over the previous one.

**Examples**

1. `fot < infile > outfile`
2. `a.out | fot | lpr`
3. `fot infile outfile`

**Diagnostics**

If the first character of a line is none of the above, the line is unchanged. Upon completion of the command, a diagnostic message is displayed in the standard error file indicating the number of lines not containing a valid Fortran carriage control character. For example:

```
invalid n lines carriage control conventions in file1
```

## 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.

## 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 `BLOCKSIZE`s and displays an elapsed time for each. You can use the results to determine an optimum value for your system to specify in your programs. Note that a particular `BLOCKSIZE` may not perform as well on other systems.

# 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:

- For dummy arguments, start each array dimension at zero (not at one, which is the default). Thus, declare an array `A` to be `A(0:99)`, not `A(100)`.
- One-dimensional arrays are more efficient than two, two are more efficient than three, etc.
- Make a direct file record length a power of two.
- Unformatted input/output is faster for numbers.
- Formatted CHARACTER input/output is faster using:

  ```
  CHARACTER*256 C
  ```

  than:

  ```
  CHARACTER*1 C(256)
  ```

## Side Effects

LF95 arguments are passed to subprograms by address, and the subprograms reference those arguments as they are defined in the called subprogram. Because of the way arguments are passed, the following side effects can result:

- Declaring a dummy argument as a different numeric data type from that declared 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 actual argument in a single CALL statement or function reference, 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. In accordance with the Fortran standard, the compiler and/or runtime is not required to notice such changes; this allows optimizations to be performed (e.g., keeping the second dummy argument, or elements thereof, in registers).
- 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 maximum length may be assigned in the OPEN statement `RECL=` specifier. A linefeed character 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 linefeed.

### 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 maximum length may be assigned in the OPEN statement `RECL=` specifier. The beginning area is used when an unformatted sequential READ statement is executed. The end area is used when a BACKSPACE statement is executed.

## Direct File Format (Formatted)

Files processed by formatted direct input/output statements have a fixed length record format. One Fortran record corresponds to one logical record. The length of the logical record must be assigned in the OPEN statement `RECL=` specifier. If the Fortran record is shorter than the logical record, the remaining part is padded with blanks. The length of the Fortran record must not exceed the logical record. This fixed length record format is unique to Fortran.

## Direct File Format (Unformatted)

Files processed by unformatted direct-access 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.

## Binary File Format

Files opened with `FORM='BINARY'` (or `ACCESS='TRANSPARENT'`) are processed as a stream of bytes with no record separators. While any format of file can be processed as binary, you must know its format to process it correctly. Note that, even though `ACCESS='TRANSPARENT'` is supported by LF95, `FORM='BINARY'` is the preferred method of opening such files. Note that these specifiers are not currently part of the Fortran standard and may vary from one compiler to the next; however, this may change in future versions of the Fortran standard.

## Endfile Records

An endfile record must be the last record of a sequential file. Endfile records do not have a length attribute. The ENDFILE statement writes an endfile record in a sequential file. After at least one WRITE statement is executed, an endfile record is output under the following conditions:

- A REWIND statement is executed.
- A BACKSPACE statement is executed.
- A CLOSE statement is executed.

## Porting Unformatted Files

Unformatted files created on other platforms can be accommodated with certain runtime options. “Big-endian” numeric data (integer, logical, and IEEE floating-point) can be accommodated with runtime option T. Note that the big-endian conversion is not performed for real variables that are elements of a derived type if the whole type is being read. IBM370-format floating-point data can be accommodated with runtime options C and M (see *“Runtime Options”* on page 115). By default, LF95 reads and writes numeric data in “little-endian” format.

## File Creation: Default Names

If a file is opened without specifying a filename, the file is assigned the name `fort.`*unit*, where *unit* is the unit number specified in the OPEN statement.

If a file is opened as `STATUS='SCRATCH'`, and `FILE=` is not specified, then the file is assigned a random name and is created in the system temporary directory. If `FILE=` is specified, then the file is created in the current working directory. In both cases, the file is deleted upon program termination, even if it is closed with `STATUS='KEEP'` (see *“Intermediate Files”* on page 11).

Normal program termination causes all files to be closed.

## Link Time

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
```

## 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 117.

## Limits of Operation

**Table 11: LF95 Limits of Operation**

| Item | Maximum |
|---|---|
| program size | 4 Gigabytes or available memory (including virtual memory), whichever is smaller |
| number of files open concurrently | system dependent (see `limits` command of `csh`; subtract three for Fortran units 0, 5, and 6 from the system limit) |
| Length of CHARACTER datum | 2,147,418,072 bytes |
| I/O block size | 2,147,483,647 bytes |
| I/O record length | 2,147,483,647 bytes |
| I/O file size | 18,446,744,073,709,551,614 bytes |
| I/O maximum number of records (direct-access files) | 18,446,744,073,709,551,614 divided by the value of the RECL= specifier |
| nesting depth of function, array section, array element, and sub-string references | 255 |
| nesting depth of DO, CASE, and IF statements | 50 |
| nesting depth of implied-DO loops | 25 |
| nesting depth of INCLUDE files | 16 |
| number of array dimensions | 7 |
| array size | *T*, where 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: $T = l_1 \times s + \sum_{i=2}^{n} \left\{ l_i \times \left( \prod_{m=2}^{i} d_m - 1 \times s \right) \right\}$ *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 execution time 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 the behavior of input/output operations, diagnostic reporting, and floating-point operations.

Runtime options submitted on the command line are specified by using a character sequence that uniquely identifies the runtime options, so that they may be distinguished from regular command line arguments utilized by the user's program. In the current version of the compiler, the values obtained via the GETCL(), GETPARM(), and GETARG() functions will include the runtime options as well as user-defined command line arguments. This can cause problems if the number of runtime options specified is always changing or is unknown to the programmer. The solution in this case is to place the runtime options in environment variable FORT90L (see *"Environment Variables"* on page 116).

## 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," 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.

## Environment Variables

As an alternative to the command line, the environment 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:

```
setenv FORT90L=-Wl,-e99,-le
a.out -Wl,-m99 -myopt
```

has the same effect as the command line

```
a.out -Wl,-e99,-le,-m99 -myopt
```

The result is that when executing the program `a.out`, the runtime options -`e99`, `-le`, `-m99`, and user-defined executable program argument `-myopt` are in effect.

**Example 2:**
When the following command lines are used,

```
setenv FORT90L=-Wl,-e10
a.out -Wl,-e99
```

the result is that a.out is executed with runtime option `-e99` in effect, overriding the option `-e10` set by shell variable FORT90L.

Note that `setenv` would be `export` in the examples above for Korn and bash shell users.

## 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 12: 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, Output, and Error

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`*[,option][,option]...*

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. No space is allowed after a comma. If the same runtime option is specified more than once, the last occurrence is used.

**Example:**

```
a.out -Wl,-a,-p10,-x
```

## Descriptions of Runtime Options

### -C or -C[*u_no*]

#### Convert IBM370 Floating Point Format

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 REAL and DOUBLE PRECISION 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 *u_no* specifies an integer from 0 to 2147483647 as the unit number. If optional argument *u_no* 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 *u_no* is regarded as IEEE-format floating-point data in an unformatted input-output statement.

**Example:**

```
a.out -Wl,-C10
```

### -M

#### Mantissa Conversion Error Reporting for IBM370 data

The `-M` option specifies whether to output the diagnostic message (0147i-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 (0147i-w) is not output.

**Example:**

```
a.out -Wl,-M
```

### -Q

#### Blank-padding for Formatted Input

The `-Q` option suppresses padding of an input field with blanks when a formatted input statement is used to read a Fortran record (this behavior will apply to all unit numbers). This option applies to cases where the field width needed in a formatted input statement is longer than the length of the Fortran record and the file was not opened with an OPEN statement.

The result is the same as if the PAD= specifier in an OPEN statement is set to NO. If the `-Q` option is omitted, the input record is padded with blanks. The result is the same as when the PAD= specifier in an OPEN statement is set to YES or when the PAD= specifier is omitted.

**Example:**

```
a.out -Wl,-Q
```

## -Re

**Runtime Error Handling**

Disables the runtime error handler. Traceback, error summaries, user control of errors by service routines ERRSET and ERRSAV, and execution of user code for error correction are suppressed. If possible, the standard correction will be performed when an error occurs.

**Example:**

```
a.out -Wl,-Re
```

## -Rm:*filename*

**Runtime Diagnostic Output to File**

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.out -Wl,-Rm:errors.txt
```

## -Ry

**Y2K (Year 2000) Compliance Diagnostics**

Encourages Y2K compliance at runtime by generating an i-level (information) diagnostic whenever code is encountered which may cause problems after the year 2000 A.D. Must be used in conjunction with the `-li` option in order to view diagnostic output.

**Example:**

```
a.out -Wl,-Ry,-li
```

## -T or -T[*unit*]

**Big-endian Data Conversion**

“Big-endian” data (integer, logical, and IEEE floating-point) is transferred in an unformatted input/output statement. The optional argument *unit* is a unit number, valued between 0 and 2147483647, connected with an unformatted file. If *unit* is omitted, `-T` takes effect for all unit numbers. If both `-T` and `-T`*unit* are specified, then `-T` takes effect for all unit numbers. By default, LF95 reads and writes numeric data (integer, logical, and IEEE floating-point) as “little-endian.” Note that this conversion is not performed if the real variable is a component of a derived type, and the whole type is being read.

**Example:**

```
    a.out -Wl,-T10
```

### -a

#### Force Abnormal Termination

When the `-a option` is specified, an abend (abnormal termination event) is forcibly executed following normal program termination. This processing is executed immediately before closing external files.

**Example:**

```
    a.out -Wl,-a
```

### -d[*num*]    1 ≤ *num* ≤ 32767

#### Direct Access I/O Work Area

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 are 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`*unit*`bf` where *unit* is the unit number (see *"Shell Variables for Input/Output"* on page 123 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 one. If the `-d` option is omitted, the size of the input/output work area is four kilobytes.

**Example:**

```
    a.out -Wl,-d8
```

### -e[*num*]   0 ≤ *num* ≤ 32767

#### Execution error limit

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.out -Wl,e10
```

### -g[*num*] 1 ≤ *num*

**Sequential Access I/O Work Area**

The `-g` option sets the size of the input/output work area used by sequential access input/output statements. 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 one or more. If the `-g` option is omitted, the size of the input/output work area defaults to eight kilobytes.

The `-g` option improves input/output performance when a large amount of data are 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`*unit*`bf`, where *unit* is the unit number (see *"Shell Variables for Input/Output"* on page 123 for details). When the `-g` option is specified at the same time as the shell variable `fu`*unit*`bf`, the `-g` option has precedence.

**Example:**

```
       a.out -Wl,-g10
```

### -i

**Interrupt Processing**

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.out -Wl,-i
```

### -l*errlevel* *errlevel*: { i | w | e | s }

**Diagnostic Reporting Level**

The `-l` option (lowercase L) controls the output of diagnostic messages during execution. The optional argument *errlevel*, 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.out -Wl,-le
```

### -m*unit*   0 ≤ *unit* ≤ 2147483647

**Standard Error Output**

The `-m` option connects the specified unit number *unit* to the standard error output file/device (STDERR) where diagnostic messages are to be written. Argument *unit* 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. Care should be taken to avoid conflict with units specified by `-p` and `-r` options. Also, see *"Shell Variables for Input/Output"* on page 123 for further details.

**Example:**

```
a.out -Wl,-m10
```

### -n

**Prompt Messages, Standard Input**

The `-n` option controls whether prompt messages are sent to standard input (STDIN). When the `-n` option is specified, prompt messages are output when data are 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 are to be entered from standard input using a formatted sequential READ statement.

**Example:**

```
a.out -Wl,-n
```

### -p*unit*   0 ≤ *unit* ≤ 2147483647

**Standard Output**

The p option connects the unit number *unit* to the standard output file/device (STDOUT), where *unit* 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. Care should be taken to avoid conflict with units specified by `-m` and `-r` options. Also, see *"Shell Variables for Input/Output"* on page 123 for further details.

**Example:**

```
a.out -Wl,-p10
```

### -q

**Capitalize Numeric Edit Output Characters**

The `-q` option specifies whether to capitalize the E, EN, ES, D, Q, G, L, and Z numeric 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.

**Example:**

```
a.out -Wl,-q
```

### -r*unit*     0 ≤ *unit* ≤ 2147483647

**Standard Input**

The `-r` option connects the unit number *unit* to the standard input file/device (STDIN) during execution, where *unit* 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. Care should be taken to avoid conflict with units specified by `-m` and `-p` options. Also, see *"Shell Variables for Input/Output"* on page 123 for further details.

**Example:**

```
a.out -Wl,-r10
```

### -u

**Underflow Interrupt Processing**

The `-u` option controls floating point underflow interrupt processing. If the `-u` option is specified, LF95 performs floating point underflow interrupt processing. The system may output diagnostic message0012i-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.out -Wl,-u
```

### -x

**Blanks in Numeric Formatted Input**

The `-x` option determines whether blanks in numeric formatted 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.out -Wl,-x
```

## Shell Variables for Input/Output

This section describes shell variables that control file input/output operations. These environment variables are lower-case unless otherwise indicated.

### **fu*unit* = *filename*** **00 ≤ *unit* ≤ 2147483647**

The `fu`*unit* shell variable pre-connects units to files. The value *unit* is a unit number (must be at least two digits). The value *filename* is a file to be connected to unit number *unit*. The standard input and output files (`fu05` and `fu06`) and error file (`fu00`) must be avoided, unless their values have been modified using the -m, -p, or -r options, in which case those new values must be avoided.

The following example shows how to connect myfile.dat to unit number 10 prior to the start of execution.

**Example:**

```
setenv fu10 myfile.dat
```

### **fu*unit*bf *size*** **00 ≤ *unit* ≤ 2147483647**

The `fu`*unit*`bf` shell variable specifies the size of the input/output work area used by sequential or direct access input/output statements. This applies equally to both formatted and unformatted files. The value *unit* in the `fu`*unit*`bf` shell variable specifies the unit number (the number must have at least two digits). 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 specified for one unit does not automatically apply to other units.

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 eight 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 four kilobytes.

**Example 1:**

Sequential Access Input/Output Statements.

```
setenv fu10bf 64
```

When sequential access input/output statements are executed for unit number 10, the statements use an input/output work area of 64 kilobytes.

**Example 2:**

Direct Access Input/Output Statements.

```
setenv fu10bf 50
```

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.

# C Lahey Technical Support

Lahey Computer Systems takes pride in the relationships we have with our customers. We maintain these relationships by providing quality technical support, an informative website, newsletters, product brochures, and new release announcements. In addition, we listen carefully to your comments and suggestions. The World Wide Web site has product patch files, new Lahey product announcements, lists of Lahey-compatible software vendors and information about downloading other Fortran-related software.

## Hours

### Lahey's Business Hours Are

7:45 A.M. to 5:00 P.M. Pacific Time Monday - Thursday

7:45 A.M. to 12:45 P.M. Pacific Time Friday

### Telephone Technical Support Is Available

8:30 A.M. to 3:30 P.M. Pacific Time Monday - Thursday

8:30 A.M. to 12:00 P.M. Pacific Time Friday

### We Have Several Ways for You to Communicate with Us:

| | |
|---|---|
| • PHONE: | (775) 831-2500 |
| • FAX: | (775) 831-8123 |
| • E-MAIL: | support@lahey.com |
| • ADDRESS: | 865 Tahoe Blvd. |
| | P.O. Box 6091 |
| | Incline Village, NV  89450-6091 U.S.A. |
| • WWW: | http://www.lahey.com |

## Technical Support Services

Lahey provides free technical support to registered users of current versions of our products. This support is available by e-mail, fax, and mail for all products. For LF95 PRO, technical support is also available by telephone. Technical support includes assistance in the use of our software and in getting any bugs you may find in our software fixed. It does not include tutoring in how to program in Fortran or how to use any host operating system or operating system APIs.

### How Lahey Fixes Bugs

Lahey's technical support goal is to make sure you can create working executables using LF95. Towards this end, Lahey maintains a bug reporting and prioritized resolution system. We give a bug a priority based on its severity.

The definition of any bug's severity is determined by whether or not it directly affects your ability to build and execute a program. If a bug keeps you from being able to build or execute your program, it receives the highest priority. If you report a bug that does not keep you from creating a working program, it receives a lower priority. Also, if Lahey can provide a "workaround" to the bug, it receives a lower priority.

In recognizing that problems sometimes occur in changing software versions, Lahey allows you to revert to an earlier version of the software until Lahey resolves the problem.

### Contacting Lahey

To expedite support services, we prefer written or electronic communications via fax or e-mail. These systems receive higher priority service and minimize the chances for any mistakes in our communications.

Before contacting Lahey Technical Support, we suggest you do the following to help us process your report.

- Determine if the problem is specific to code you created. Can you reproduce it using the demo programs we provide?
- If you have another machine available, does the problem occur on it?

### Information You Provide

When contacting Lahey, please include or have available the information listed below.

- Registered user name
- Registered serial number
- Product title and version (for example, LF95 v5.5)
- Patch level (for example, the h patch)
- Operating system (for example, Windows 98 or Redhat Linux v6.0)

- A short source code example. This will allow us to reproduce the problem. Please make sure the source code is as short as possible to allow us to analyze your issue quickly. Attach the source code file to your e-mail to support@lahey.com.
- Third-party products used. If you are using an add-on library (such as Winteracter) or productivity tool (such as Visual Analyzer), provide the name and version of this product. If your application is mixed-language (such as Fortran and C), provide the name and version of the non-Fortran language system.
- System environment settings

  To save your environment variables in a text file, go to a command prompt and redirect the output of the SET command to a file:

  ```
  SET > SETCMD.OUT
  ```

  Attach the `SETCMD.OUT` file to your e-mail to support@lahey.com.
- Step-by-step problem description. Tell us the sequence of commands or buttons used that lead up to the problem occurring. Remember, if we can't reproduce it, we can't fix it for you.
- Compiler, linker, or Make/Automake messages.
- While simply typing the complete error message is always an option, you can save extensive messages to a text file to send to us, if that is easier. To save the messages as a text file, from the command line redirect the command output as in the following example:

  ```
  your_command_line > CMD.OUT
  ```

- Attach the `CMD.OUT` file to your e-mail to support@lahey.com. If you are using the ED editor, run your compile command and attach the ERRS.* file of the working directory to your e-mail to support@lahey.com
- Exact text of error message or Window message box.

Support is provided free to solve problems with our products, and to answer questions on how to use Lahey products. Support personnel are not available to teach programming, debug programs, or answer questions about the use of non-Lahey products or tools (such as MS Windows, Linux, MS Visual Basic, etc.). These services are provided on a paid consulting basis.

## World Wide Web Site

Our URL is `http://www.lahey.com`. Visit our web site to get the latest information and product patch files and to access other sites of interest to Fortran programmers.

## Lahey Warranties

### Lahey's 30 Day Money Back Guarantee

Lahey agrees to unconditionally refund to the purchaser the entire purchase price of the product (including shipping charges up to a maximum of $10.00) within 30 days of the original purchase date.

All refunds require a Lahey Returned Materials Authorization (RMA) number. Lahey must receive the returned product within 15 days of assigning you an RMA number. If you purchased your Lahey product through a software dealer, the return must be negotiated through that dealer.

### Lahey's Extended Warranty

Lahey agrees to refund to the purchaser the entire purchase price of the product (excluding shipping) at any time subject to the conditions stated below.

All refunds require a Lahey Returned Materials Authorization (RMA) number. Lahey must receive the returned product in good condition within 15 days of assigning you an RMA number.

You may return an LF95 Language System if:

- It is determined not to be a full implementation of the Fortran 95 Standard and Lahey does not fix the deviation from the standard within 60 days of your report.
- Lahey fails to fix a bug with the highest priority within 60 days of verifying your report.
- All returns following the original 60 days of ownership are subject to Lahey's discretion. If Lahey has provided you with a source code workaround, a compiler patch, a new library, or a reassembled compiler within 60 days of verifying your bug report, the problem is considered by Lahey to be solved and no product return and refund is considered justified.

## Return Procedure

You must report the reason for the refund request to a Lahey Solutions Representative and receive an RMA number. This RMA number must be clearly visible on the outside of the return shipping carton. Lahey must receive the returned product within 15 days of assigning you an RMA number. You must destroy the following files before returning the product for a refund:

- All copies of Lahey files delivered to you on the software disks and all backup copies.
- All files created by this Lahey Language System.

A signed statement of compliance to the conditions listed above must be included with the returned software. Copy the following example for this statement of compliance:

I, ________________________________________(your name), in accordance with the terms specified here, acknowledge that I have destroyed all backup copies of and all other files created with the Lahey software. I no longer have in my possession any copies of the returned files or documentation. Any violation of this agreement will bring legal action governed by the laws of the State of Nevada.

Signature:

Print Name:

Company Name:

Address:

Telephone:

| | | |
|---|---|---|
| Product: | Version: | Serial #: |

RMA Number:

Refund Check Payable To:

## Return Shipping Instructions

You must package the software diskettes with the manual and write the RMA number on the outside of the shipping carton. Shipping charges incurred will not be reimbursed. Ship to:

Lahey Computer Systems, Inc.
865 Tahoe Blvd.
P.O. Box 6091
Incline Village, NV 89450-6091
U.S.A.

# Index