

Chapter 15: Software Development

This chapter was initially written in 1996. We checked the contents (2003) to make sure they are still valid, and have updated the existing information as necessary, in particular, including Linux specifics. However, we have not added information such that the chapter reflects the state of software development in 2003. We are keeping this chapter in *UNIX at Fermilab* because it is valid if not current, and it may be helpful to some users.

The various experiments and projects at Fermilab have their own environments for code management and development, in most cases. Check with the software managers in your experiment or group to find out what tools you're expected to use, what standards you're supposed to follow, and so on.

This chapter gives an introduction to UNIX software development tools in common use at Fermilab, providing information on:

- Supported languages
- Compiling and linking in C, C++ and FORTRAN
- Debugging

We do not include a discussion of general programming here, but rather, aspects of software development particular to UNIX. You will need to reference the man pages and the vendors' manuals for more system-specific details.

Useful documents relating to FORTRAN, C, and C++ programming can be found under **Software Development** on the *UNIX Resources* Web page.

Many software development subjects of interest to Fermilab users are beyond the scope of this manual. Among them are:

- Object Oriented programming techniques including NextStep
- CASE tools
- Neural Network methods
- Lattice Gauge techniques used in ACP-MAPS, CANOPY, et.al.
- Data Mining methods available in the CAP facility.

- Many excellent commercial tools for building Graphics User Interfaces, debugging, migrating and analyzing performance of user code, building Database interfaces, etc.

15.1 Overview of Programming Languages and Tools

This is a short overview of some common programming languages. Our aim here is to give you a general idea of what tools are available, and how they can be used. The list is ordered from low to high level, and indicates common uses:

Assembler	not used at Fermilab
C	system services, user interface, general utilities
FORTRAN	FORmula TRANslation (physics calculations)
C++	object oriented general programming
Perl	interpreter, general purpose
Python	object-oriented and/or general purpose interpreted scripting language
Tk	interpreter, GUI interfaces

Assembler

Traditionally, assembler has been needed for a couple of reasons:

- Access to system services and hardware
- Tuning program efficiency

Assembler programming is not generally necessary or desirable on the RISC based UNIX systems presently in use. RISC system performance is so heavily dependent on pipelining and various caches that it is extremely difficult, if not impossible, for an individual to write more efficient assembler code than is generated by the higher level language compilers. The C language provides all the hardware access and system service capabilities traditionally provided by Assembler.

C

As noted above, C has filled the programming niche traditionally occupied by Assembly language. In addition to its normal use as a high-level programming language, C can act as a universally portable Assembler.

Because the C language was created, has evolved, and has become standardized hand-in-hand with the UNIX operating system, it is the language of choice for applications involving system services and user interfaces. Using C is discussed in several of the following sections.

Now that an ANSI C standard exists and is widely implemented, portability of C code is much improved. ANSI C compilers are the default on most Fermilab systems. Likewise, adoption and availability of POSIX standards for operating system services has greatly improved program portability.

An excellent reference book for C programming is *The C Programming Language* by Kernighan and Ritchie published by Prentice-Hall.

FORTRAN

FORTRAN remains the most effective language for mathematical calculations. This is due partly to decades of research which has produced highly efficient optimizing compilers, and partly to the millions of lines of tested, portable code already in use.

C++

C++ adds object oriented programming constructs to the C language. At the risk of oversimplifying, it seems that C++ is substantially harder to learn and to use for writing new programs, but the resulting programs are much better structured, more maintainable, and more shareable than traditional C or FORTRAN programs.

An additional advantage of C++ is the availability of class¹ libraries. A Standard Template Library will come with most compilers soon. This library will contain many useful low level classes for such things as strings and streams I/O. Also, some vendors include other commercial class libraries as added value to their compilers. In addition to the vendor-supplied versions, C++ is available as a part of Gnu C. This has increased its popularity.

The C++ language standard is (still!) in the process of adoption by the ANSI and ISO standards committees. Since the C++ standard has not been finalized (although it changed significantly in December 1996), there are at this time (November 1997) cross-platform porting issues. None of our major vendors (SGI, IBM, GNU) are yet providing a draft-standard-compliant compiling and runtime toolkit. The safest bet for generating portable C++ code is to avoid using newer features such as exceptions and templates, which may be implemented differently (or not at all) by the various C++ compiler vendors, at least until the vendors catch up with the standard. As an alternative, use the `g++` command to run the Gnu integrated C/C++ compiler on all platforms. Using the `g++` command instead of `gcc` gives you appropriate C++ linking.

1. A *class* is similar to a *structure definition* in C.

For documentation, you may refer to:

- the **CC** (upper case), **gcc** and **g++** command man pages
- *The C++ Programming Language*, Addison-Wesley, by Bjarne Stroustrup

Perl

perl is installed at Fermilab as part of the **shells** product. The man page for **perl** gives a good brief description:

perl is an interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It's also a good language for many system management tasks. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal).

There is an excellent text published by O'Reilly & Associates, Inc. on **perl**.

Python

Python is an interpreted, interactive, object-oriented programming language often compared to **Tcl**, **Perl**, **Scheme** and **Java**. **Python** combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries, as well as to various windowing systems (e.g., **X11**, **Motif**, **Tk**, **Mac**, **MFC**, **STDWIN**). New built-in modules are easily written in C or C++. Python is also usable as an extension language for applications that require a programmable interface.

An on-line document for **Python** is available in the CD documentation database.

Tk

Tk is an **X11** toolkit that provides the Motif look and feel, and is easy to use for building graphical interfaces largely because it is built on an interpreted language. It can be used with a variety of languages including **Tcl** (**Tk** used to be *solely* implemented using **Tcl**), **Perl**, and **Python**.

The best reference for **Tk** is the book *Tcl and the Tk Toolkit*¹, by John K. Ousterhout, published by Addison-Wesley. The `README` file under the `tk` directory in `$TK_DIR` (created during setup) points to a draft of this book. Also see the man pages for information on these languages.

1. Some publishers' catalogues use an ampersand (&) rather than the word "and"; check both in database searches.

Other Languages and Language-Related Tools

Other tools exist that are commonly used as languages in the appropriate context. These include, for example, the various UNIX shells (as discussed in section 5.4 *Shell Scripts*), and **awk**, **sed**, **yacc**, and **lex**, for which O'Reilly & Associates, Inc. provides excellent texts.

There are many other languages which are not widely supported, are supported on a per-project basis, or are not in general use at Fermilab. Among these are Java, Pascal, Modula-2, Lisp, Forth, and Bliss. We do not discuss them in this document.

15.2 Introduction to C and FORTRAN on UNIX

15.2.1 The C Compiler: **cc**

cc (lower case) is the vendor-supplied C compiler command on all Fermilab-supported UNIX systems (except LINUX, where it is **gcc**). ANSI C compilers are the default on most Fermilab systems. To compile one or more C source files (`<filename>.c`), you run the **cc** utility. **cc** automatically invokes the link editor **ld** unless the option **-c**, which explicitly suppresses linking, is used.

15.2.2 The FORTRAN Compiler: **f77**

All the Fermilab supported UNIX systems have good FORTRAN 77 compilers, which provide some minimal extensions. These compilers also recognize most DEC-supported VAX-FORTRAN extensions.

f77 is the FORTRAN compiler command on all Fermilab-supported UNIX systems (on LINUX, **f77** actually runs **g77**). The **f77** command controls both compilation and linking functions automatically using appropriate FORTRAN runtime libraries. **f77** can produce object modules, partially linked objects, or executable programs, as appropriate. The option **-c** explicitly suppresses linking.

15.2.3 C and FORTRAN Compiling Basics

UNIX compilers, including **f77** and **cc**, generally use both filename extensions and the file content to determine how to handle the files listed on the command line. The commonly used extensions are:

For C

`c` C source, e.g., `myfile.c`

For FORTRAN

`f` FORTRAN source, e.g., `myfile.f`

For both

`o` object file, e.g., `myfile.o`

`a` archive library, e.g., `mylib.a`

(none) executable image, e.g., `myfile`

For historical reasons the UNIX linkers produce by default an executable named `a.out`; the option `-o <filename>` is available to override this default.

We list here some basic compiling and linking examples. In these examples, we use a source file named `foo.c`, where `c` is the standard extension for C. In all instances shown here `c` can be directly replaced by `f`, and `cc` by `f77`, for FORTRAN.

To produce the `foo.o` object module, enter the command:

```
% cc -c foo.c
```

To produce the `foo` executable from source, enter the command:

```
% cc -o foo foo.c
```

To produce the `foo` executable from source + object file, enter the command:

```
% cc -o foo foo.c myobj.o
```

To produce the `foo` executable from source + library, enter the command:

```
% cc -o foo foo.c $CERN_DIR/lib/libmathlib.a
```

To produce the `foo` executable from source + X11/Motif (standard system libraries), enter the command:

```
% cc -o foo foo.c -lXm -lXt -lX11
```

The options used with the `cc` and `f77` commands are discussed in later sections.

15.2.4 Linking Order

Most UNIX linkers process source, object and library files in the order that they occur on the command line. Backward library references, from a file to an earlier library, will not be satisfied. It may be necessary to list a library more than once for successful linking.

15.2.5 Displaying Active Options

You may wish to know which options are active for a given compilation, in order to verify that the defaults are what you expect. Each platform seems to provide this information somewhat differently:

Platform	Option	Result
Linux	<code>-v</code>	Options and other details, to <code>stdout</code>
IRIX	<code>-v</code>	Options and other details, to <code>stdout</code>
SunOS	<code>-'#'</code>	Options and other details, to <code>stdout</code>

15.2.6 Option Passing

At each stage of compilation, any unrecognized command line options are passed on to the next stage. Options that could be valid for more than one stage can be explicitly passed to a particular stage. To direct an option to a specific phase of compilation or linking, use the option `-w` (for IRIX) or `-Qoption` (for SunOS). The phase is identified by the letter immediately following the `-w`. Thus, for example, `-wl, -m` tells `f77` or `cc` to pass the option string `-m` to the `l` (link) phase. The `f77` command line might read:

```
% f77 -wl, -m foo.f for IRIX, and
```

```
% f77 -Qoption ld -m foo.f
```

for SunOS (where `ld` is the loader program)

15.3 Introduction to C++ on UNIX

The bigger experiments have a large infrastructure around their C++ code development. Users should refer to their experiment-specific websites for development information, tutorials, and so on.

CC (upper case), **g++** and **gcc** are all C++ compiler commands on the Fermilab-supported UNIX systems that provide C++ (see section 15.1 *Overview of Programming Languages and Tools* for a brief discussion). Just as C++ is a superset of C, the C++ compilers are very similar to C compilers in that their options are usually a superset of C compiler options. The basic compiling information about C in section 15.2.3 *C and FORTRAN Compiling Basics* is also applicable to C++, with the following exceptions¹:

- Source filename extension conventions are compiler-dependent. Check the man pages for **CC** (upper case) to determine the extensions used on your system. Extensions include:

<code>C</code>	(upper case)
<code>c</code>	(lower case)
<code>cxx</code>	
<code>cpp</code>	
<code>cc</code>	
<code>c++</code>	
<code>hh</code>	c++ header file
<code>chh</code>	c++ header file
<code>icc</code>	c++ included source file (usually inline function definitions).

- The C++ compiler is invoked with **CC** (upper-case), **g++** or **gcc**
- There are additional compiler options specific to C++; see the man pages.

The C++ compiler may not yet be installed on your Fermilab UNIX system.

15.4 C, C++ and FORTRAN Compiler Options

The default compiler options will produce a usable program on any of the supported platforms, however they may not be optimal for many situations. Several additional options are discussed in this section.

A caveat: One very annoying “feature” of the **f77** and **cc** commands is that some of the options must be specified with whitespace (at least one blank character) between the option identifier and the option value, others without whitespace and still others with or without, according to the user’s choice. For

1. This doesn’t apply to Gnu C++. The compiler command for both Gnu C and Gnu C++ is **gcc**, and the available Gnu compiler options are different from the vendor compiler options.

instance, on IRIX systems, there must be whitespace between `-o` and the name of the executable file but there must not be any whitespace between `-l` and the library file name.

If compilers are upgraded it is possible that some of these options could change. For full details on all the options, see the man pages and the vendor compiler documentation.

15.4.1 Commonly-Used Options

These options are valid for C, C++ and Fortran:

- `-c` suppress linking, produce object file `*.o`. The linker is called as part of the compilation process by default.
- `-L <directory>` add `<directory>` to the default linker search list; not needed for user libraries. The `-L` option adds directories to the linker's default search path. `-L` directories are searched ahead of system library areas. A user library could accidentally match the name of an obscure system library, with startling results.
- `-l<file>` search library `lib<file>.a`, from the default areas (the `lib` gets prefixed and the `.a` appended automatically)

With user-written libraries, specify the libraries on the command line with their true file names and a full path, without using the `-L` and `-l` options.

- `-o <file>` (lower case `o`) produce executable program `<file>` rather than `a.out`
- `-O<n>` (upper case `O`) optimize at level `<n>` where `<n>` is 0, 1, 2, 3, or 4. The meanings of the different optimization levels vary from system to system. See the man pages for details.
- `-w` suppress informational and non-fatal compiler warnings
- `-P` (upper case `P`) run `cpp` (C preprocessor) only to produce `*.i` source listing
- `-p` (lower case `p`) enable profiling (used with the `prof` profiler); see the man pages for details
- `-gp` enable profiling (used with the `gprof` profiler); see the man pages for details

Other commonly-used C and C++ compiler options are:

- `-I <directory_name>` extend include path
- `-D` set value of preprocessor macro

Other commonly-used FORTRAN compiler options are:

- u** IMPLICIT NONE
- C** (upper case **C**) check runtime subscript range

15.4.2 Recommended Options for General Use

As mentioned earlier, the default options may not be optimal for a number of very common situations, namely for debugging, moving binary code between non-identical machines, or tuning for best performance. Even for general usage, some non-default options have proven helpful in avoiding internal compiler limits and providing better compatibility for migrated code.

This list shows options by platform which apply to all situations, and which we recommend for general use. They can be used in addition to other options you might choose based on your specific needs. As stated above, if compilers are upgraded it is possible that some of these options could change, so you should always consult the man pages.

- IRIX **-trapuv -lfpe**
- SunOS **-xl -fnonstd**
- Linux

Discussion of General Options

- trapuv** (IRIX) set uninitialized variables to NaN, to help catch nonportable code and latent bugs at runtime.
- lfpe** (IRIX) use the Floating Point Exception library, to report or core dump on errors. Without **-lfpe**, errors produce NaN values silently. Must be combined with the `TRAP_FPE` environment variable, or calls to `handle_fpes`, to be effective. `TRAP_FPE` must be set to the value:

`OVERFL=ABORT ; DIVZERO=ABORT ; INVALID=ABORT`

See section 9.5 *Shell Variables and Environment Variables* for setting environment variables.
- xl** (SunOS) Extended Language, for Fermilab-required extensions (see section 15.2.2)
- fnonstd** (SunOS) trap floating point errors.

15.4.3 Debugging Option

-g	include full symbol table for debugger. This option interacts with the optimizer differently on each platform, so we provide some usage notes:
IRIX	-g forces optimization off; use -g3 to permit optimization
SunOS	you should use -O1 (upper case letter o, and number 1) with -g

15.4.4 ABI Options Under IRIX 6

Under IRIX 6, there are three Application Binary Interface (ABI) options available. The selected option specifies at compile time which subset of the processor chip's capabilities is to be used. In previous releases of IRIX, and in all releases of the other supported UNIX flavors (as of this writing), the user is not given a choice of ABI. Under IRIX 6, choosing an ABI is generally necessary if you're doing mixed-language programming or using any libraries other than those supplied by the vendor. The available ABIs and their associated command line options are:

O32 (-32)	conforms to the ABI used with all prior IRIX releases
N64 (-64)	puts the processor into full 64-bit mode
N32 (-n32)	leaves the processor in 32-bit mode but takes advantage of a number of newer features and generally produces more efficient code on the newer processors



This topic is discussed in the Web page *IRIX 6 Application Binary Interfaces*, available from the CD home page; follow the Computational Physics link and look under *Cern Library at Fermilab*.

15.4.5 Speed Optimization Options

Note that on most platforms a combination of options is required for best optimization. Recall that these options may be used in addition to general recommended and other options. The **-O**'s here are all upper case letter o's.

IRIX	-O3 -mips2
SunOS	-O3 -cg92 -libmil

Discussion of Speed Optimization Options

Platform	Options	Comment
IRIX	-mips2	Use full R4000 instruction set. This is important on R4000 and later systems.
SunOS	-cg92 -libm il	SuperSPARC V8 instruction set Sparcstation 2 and later. See the man fpversion document. Hardware-specific inline math

15.4.6 Load Map Option

The load map option is actually a linker option. On IRIX this requires passing to the link phase the option that controls production of a load map.

Platform	Option	Output
IRIX	-wl, -m	stdout (SysV style list of input/output)
IRIX	-wl, -M	stdout (BSD style primitive map)
SunOS	-m	stdout

15.4.7 Special FORTRAN Compiler Options

Source Code Listing Option

Each platform has an option that produces a source code listing. The file extensions for these listings are platform-dependent.

Platform	Option	Listing Extension
IRIX	-listing	L
SunOS	-xlist	lst

15.5 FORTRAN Programming

There is some additional information about using FORTRAN in the UNIX environment that you will find useful to know.

15.5.1 External Reference and Entry Point Names

In order to avoid conflicts with the C runtime library when FORTRAN and C programs are included in a single program, most UNIX **f77** compilers internally append an underscore to FORTRAN external references and entry point names. At Fermilab we have set up all **f77** compilers to do this by default.

15.5.2 Separate Compilation of FORTRAN Subprograms: **fsplit**

By default, most **f77** compilers pre-link all the source code being compiled, even when you specify the **-c** option. If you compile a library with a single **f77** statement, it will usually contain a single module, and be linked as a whole.

The **fsplit** utility identifies and extracts subprograms from the original FORTRAN source file into individual files in the current directory. These files can then be compiled separately so that they retain their identity when assembled into a library.

The names of the extracted individual files are taken from their corresponding subprogram names. On some systems **fsplit** will overwrite any pre-existing file, including the original source file, whose name matches any of the subprogram names.

See the man pages for more information on **fsplit**.

Linux users who want **fsplit** can find it at:

<http://rpmfind.net/linux/falsehope/home/pierre/fsplit/>.

15.5.3 Loading Block Data Modules

Many UNIX **f77** compilers enforce the standard restriction that variables in COMMON must be initialized only in BLOCKDATA subprograms.

To ensure the loading of BLOCKDATA subprograms from libraries, declare the BLOCKDATA program name as EXTERNAL in some important module which you know will be loaded.

15.5.4 Program Control

Command Line Arguments

A FORTRAN program can easily evaluate arguments included on the command line that runs the program. A couple of examples follow.

- The IARGC function returns the number of command line arguments:

```
N = IARGC ( )           Sets N to the number of command line
                        arguments
```

- The GETARG subprogram returns the value of a specified argument:

```
CALL GETARG ( I , STR ) Puts the I'th argument into string STR
```

Environment Variables

The GETENV subprogram provides the values of environment variables. For example, to copy the value of variable `MY_OUT` into string `OUTFILE`, include in your source file:

```
CALL GETENV ( 'MY_OUT' , OUTFILE )
```

Printing

The usual FORTRAN carriage control characters placed in the first column of formatted output files are not interpreted by most UNIX text handling utilities. Use the UNIX `asa` utility to convert such FORTRAN output files to an equivalent standard ASCII text form. `asa` handles blanks, 0, 1 and + in column 1, removing any other characters. See the man pages for `asa` for details.

15.5.5 Future FORTRAN Enhancements

FORTRAN 90

The FORTRAN 90 standard includes FORTRAN 77 as a subset, and makes standard many of the extensions in common use. FORTRAN 90 is not yet commonly installed at Fermilab, and in fact we recommend that you avoid using FORTRAN 90 extensions until it is widely available. This document is written for `f77` users.

15.6 Obsolete Programming Features

You may encounter these features in older code.

Calling BLOCKDATA

Some systems (VS-FORTRAN) required an explicit call to each `BLOCKDATA` routine if you wished to force loading of that routine from a library. This is not necessary on any supported UNIX system.

BUFIO

The Fermilab `bufio` product for accessing raw variable length records on tape and disk is no longer supported. Improved capabilities are being supported in `RBIO` and `DAFT`.

RANLIB (SunOS/4)

The `RANLIB` utility added necessary library symbol tables under SunOS/4. This is done automatically under SunOS 5.

ar -s Option (ULTRIX)

The `ar -s` option added necessary library symbol tables under Digital's `ULTRIX`¹ and some earlier operating systems. This is done automatically under all Fermilab supported systems.

15.7 C and FORTRAN I/O

This section mainly applies to FORTRAN. For C, all you need to know is that the `RBIO` and `DAFT` libraries mentioned below are available.

Recall that file names in UNIX are case sensitive. It is customary to use lower case for normal files, reserving upper case names like `README` for documentation and control files.

Note that you cannot use the shell metacharacter tilde (`~`) to specify a home directory within a C or FORTRAN program; `~` is valid only on a UNIX shell command line (for all shells but `sh`). `logdir` can be used within programs for this purpose (see section 7.1.2 *The Home Directory*).

1. `ULTRIX` has been superseded by Digital `UNIX` which is no longer supported at Fermilab.

15.7.1 Records

The UNIX operating system treats a disk file as a sequence of bytes. Interpretation of data as records is entirely up to individual applications. The FORTRAN I/O libraries provide the necessary record handling for FORTRAN programs. READ statements return only the content of the records, and not the control words mentioned below.

Formatted records are terminated by a new-line character <CTRL-L> (lower case L), consistent with other UNIX text handling programs.

Unformatted records are both preceded and followed by a 32 bit integer containing an exclusive byte count.

15.7.2 Tapes

Tapes of course have real physical records, and must be handled differently than disk files. A tape file is sometimes called ‘character special’ to indicate that it is not accessed on a character-by-character basis. Tape handling is covered in Chapter 14: *Data and Tape Handling*.

15.7.3 Standard Input and Output

In conformance with the FORTRAN standard, READS and WRITES to unit * are directed to `stdin` and `stdout`. You can READ and WRITE to units 5, 6, and 0 without an OPEN. They are preconnected to `stdin`, `stdout`, and `stderr`, respectively. If you OPEN and write to any other unit number # without specifying a file name, a default name of `fort.#` will be used.

15.8 Performance Tuning for C and FORTRAN

15.8.1 Optimization

Using the compiler `-O` (upper case) options can improve program execution speed by factors of 3 or more, depending on the application, over unoptimized code. Note that your libraries must also be compiled at the same level in order for this to be effective.

Beware that there are some optimizer bugs. You should always do a limited run initially with and without optimizer options, and check your answers.

For production running, use the appropriate hardware-specific optimizations for the systems running the code. These options typically tune for cache sizes, instruction sets, and other internal hardware features, resulting in sizeable speed gains. On some systems this produces an executable that will run only on the targeted architecture.

It is common practice to retain debugger symbol tables in production programs, with only a small speed penalty. You may have to exercise care that the `-g` option does not also disable optimization of such production programs. Under IRIX, you must use `-g3` to get both optimization and symbol tables.

See the suggested speed optimization options, and vendor documents for details.

Floating Point Errors

You can obtain substantial speed increases on some systems by disabling the detection and trapping of floating point errors such as overflow, division by zero, and invalid values.

On the systems with the biggest gains, this practice can produce apparently normal, but incorrect, results. For example, `1000./0.` can produce the result `1000.` It is hardly necessary to point out that this sort of thing can produce surprising physics results! For this reason our recommended options for general use are set to at least detect and report floating point errors.

Qualifiers which force precise trapping of floating point errors are generally only used when tracking down known problems, as they can impose a large performance penalty.

15.8.2 Word Length

It may be tempting to use arrays of short words to ‘save memory’. On previous generations of computers this could also speed execution. On RISC systems there is a big performance penalty for this practice.

The current generation of RISC processors are optimized for 32 and 64 bit operations. Operations on 8 bit or 16 bit words are performed several times more slowly. The processor must extract the necessary data into a longer word, perform the operation, and mask the result back into the original location.

Alignment of variables is important for the same reason. A misaligned 32 bit word requires even more shifting and masking than a 16 bit word, with an even greater performance penalty. If you must combine different length variables in a data structure such as a COMMON, place longer words earlier in the data structure.

15.8.3 Feedback

The speed of a program can be limited as much by memory access as by processor speed. Effective use of memory cache is critical to getting good performance.

Cache usage can depend on the details of the linking process. Arbitrary changes in the ordering of modules in the executable can result in nearly 20% differences in execution speed, for typical physics code. Small changes like switching between static and shared libraries, or modifying a single subroutine call in your code, can result in substantial changes in linking order and hence in performance.

For this reason, some vendors provide mechanisms for setting optimized module ordering in the executable, based on data from a trial run of the application.

15.8.4 Inlining

Many compilers provide options for replacing calls to external modules with equivalent inline code, to permit better optimization and reduce subroutine call overheads.

Physics code does not generally benefit measurably from such inlining. Inlining within a library makes the inlined modules nonreplaceable at link time, leading to confusing results and difficult debugging. In our recommended speed optimization options we stop short of the levels that introduce inlining.

15.9 C and FORTRAN Mixed Programming

It is possible, with a little care, to combine C and FORTRAN modules in the same program. Some of the issues that need attention include:

- variable types
- array indexing
- external names
- arguments
- commons
- I/O
- linking

For newly written C programs, you may wish to use the `cfortran.h` header file available in the **cern** product.



If you're programming under IRIX 6, you will need to choose an ABI. Refer to section 15.4.4 *ABI Options Under IRIX 6*.

We give here a summary of the techniques used on the Fermilab UNIX systems.

15.9.1 Variable Types

Generally, these variable types are equivalent:

FORTRAN	C
INTEGER* 1	char
INTEGER* 2	short
INTEGER* 4	int
REAL	float
REAL*8	double
LOGICAL	(unavailable)

C strings are zero-terminated, and have no intrinsic length. FORTRAN character variable lengths are given by an internal descriptor. FORTRAN character variables passed to C routines should be copied and zero-terminated before they are used.

The internal representation of FORTRAN LOGICAL variables is usually non-0/0 for .TRUE./FALSE. respectively, but it is best not to count on this.

15.9.2 Array Indexing

By default C starts indexes at 0 and FORTRAN starts them at 1. C and FORTRAN multiple index ordering is reversed. FORTRAN substring selection appears as the first C string index. See the following equivalence table:

FORTRAN	C
intv(j)	intv[j-1]

FORTRAN	C
<code>intv(j,k)</code>	<code>intv[k-1][j-1]</code>
<code>char(j)(k:k)</code>	<code>char[k-1][j-1]</code>

15.9.3 External Names

By default on Fermilab UNIX systems, the **f77** compiler modifies FORTRAN subprogram and other external names. It forces each name to lower case, and appends an underscore. Thus FORTRAN label `SUBPROG` would become C label `subprog_`.

15.9.4 Arguments

FORTRAN subprogram arguments are always passed as addresses (C pointers). C programs can specify arguments as either pointers or values. FORTRAN CHARACTER arguments are passed as pointers, followed by a set of additional values (not pointers) at the end of the argument list, giving the length of each CHARACTER argument.

C routines can always call FORTRAN routines, with due attention being given to arguments. FORTRAN routines cannot call arbitrary C routines.

15.9.5 Commons

FORTRAN COMMON's are accessible in C as *extern structs*, with the same name mapping as is used for entry points.

FORTRAN	C
<code>COMMON /FOO/ I</code>	<code>extern struct { int i ; } foo_ ;</code>
<code>K = I</code>	<code>k = foo.i ;</code>

It is best to keep your FORTRAN COMMON variables aligned on natural boundaries¹, in order to avoid potential padding words which may be inserted differently by various FORTRAN and C compilers. You get natural alignment easily by putting longer variables before shorter variables in the COMMON.

15.9.6 I/O

Mixed C/FORTRAN I/O to the same file is not advisable. Mixed C/FORTRAN I/O to `stdout`, where `stdout` is the terminal, will usually work reasonably well, making debugging easier.

15.9.7 Linking

The easiest and safest way to link C/FORTRAN programs is to use the `f77` command, which automatically includes both C and FORTRAN run time libraries. If you insist on linking with the `cc` or `ld` commands, remember to add the options:

```
-lF77 -lI77 -lm
```

15.10 Executing a Program

Once you create an executable, you run it the way you do a normal UNIX command, that is by typing its name followed by appropriate options or parameters.

You must be aware that if you have not included “dot” (`.`) in your path, whenever your executable is in a directory not explicitly included in your path, you will need to prefix the executable name with `./` to run it. This was also mentioned in section 9.6 *Some Important Variables* under *PATH*.

1. Keep all `<n>`-byte variables' addresses an exact multiple of `<n>`, for example 0, 4, 8,... for a 4-byte quantity.

15.11 Debugging

15.11.1 dbx

dbx is a utility for source-level debugging and execution of programs written in C, C++, and FORTRAN. **dbx** allows you to trace the execution of a specified object file. You can step through a program on a line-by-line basis while you examine the state of the execution environment.

Programs compiled with the **-g** option of **cc** (and other compilers) produce an object file. This object file contains symbol table information, including the names of all source files that the compiler translated to create the object file.

dbx also allows you to examine *core files* via its **where** command. A core file contains the core image produced when the object file was executed, providing information about the state of the program and the system when the failure occurred. A core file named `core` is produced by default.

dbx commands can be stored in a start-up `.dbxinit` file that resides in the current directory or in your home directory. **dbx** executes these commands just before reading the symbol table.

There are some UNIX tools which provide a more sophisticated interface to **dbx**. See your local system documentation for information on GUI-based **dbx** tools. The product **ddd** (originally an interface for **gdb**) works as a front end for **dbx** in its more recent releases. It is currently available at Fermilab as part of the **gcc** product, but we expect to release it as a separate product soon. See the *DDD User's Guide* on the Web, document number DS0230.

Running dbx

To invoke **dbx**, enter the following command:

```
% dbx [options] [<object_file> [<corefile>]]
```

where `<object_file>` is the name of the file you want to debug.

Once **dbx** is running, you should see the **(dbx)** prompt. At this point you can start issuing **dbx** commands.

Commands

There are many **dbx** commands, all described in the man pages. Some of the basic commands are **run**, **where**, **print**, **stop**, **list**, **cont**, and **quit**:

run [**<arguments>**] Begin executing the object file, passing optional command-line **<arguments>**. The arguments can include input or output redirection to a named file.

where [**<n>**] List all active functions on the stack, or only the top **<n>**.

print [**<expressions>**] Print the values of one or more comma-separated **<expressions>**. To print values of two-dimensional FORTRAN array elements use the format: **print <array_name>[1,2]**

stop **<restriction>** [**if** **<cond>**] Stop execution if specified **<restriction>** is true. Restrictions include (this is a partial list):

- at** (source line) **<n>**
- if** **<cond>**
- in** (procedure or function) **<func>**

<cond> (condition) is a Boolean expression; if it evaluates to true, then execution is stopped.

list [**<n>1** [,**<n>2**]] or **list** **<func>**

List the source text between lines **<n>1** and **<n>2**, or on lines surrounding the first statement of **<func>**. With no arguments, list the next ten lines.

cont [**at** **<n>**] [**sig** **<signal>**] Continue execution from the point at which it was stopped if no arguments. Resume from source line **<n>** or, if a **<signal>** name or number is specified, resume process as if it had received the signal.

status [**>** **<file>**] Show active **trace**, **stop**, and **when** commands

delete [**<n>**] Remove traces, stops, and whens corresponding to each command number **<n>**, given by **status**. If **<n>** is **all** remove all.

quit Exit **dbx**.

Example

You may want to start by using **dbx** to set some break points within your code. To step through your code at the very beginning, you need to stop in the `MAIN` routine if you are debugging an object file created from FORTRAN source code (stop in `main` if your source is in C language). For example, you would type:

```
(dbx) stop in MAIN
```

Now you can issue the **run** command to start execution of your object file. You will get the process id and the name of the object file being executed.

```
(dbx) run
```

At this point, you may use the **list** command for the first 10-line listing of the source code:

```
(dbx) list
```

Use the **stop** command to set break-points at various lines or procedures within the object-file:

```
(dbx) stop at 10
```

```
(dbx) stop in sub123
```

```
(dbx) stop in sub456 if i == 24
```

The execution will stop in the example above at line 10, or in subroutine `sub123`, or in subroutine `sub456` when `i` is equal to 24. To continue execution at any point in your debugging, issue the **cont** command:

```
(dbx) cont
```

To restart your debugging session, issue the **rerun** command:

```
(dbx) rerun
```

To exit **dbx**, type **quit**:

```
(dbx) quit
```

Usage Note

A user reports that when using **dbx <object_file> core** he has found it useful to turn on all but one of the IEEE arithmetic traps, in order to stop execution when the arithmetic fault occurs (instead of continuing with some default action and then reporting that the following IEEE arithmetic flags had been set). He located a spurious division by zero in this manner. The **f77** man page for Solaris describes the necessary flag value on the **f77** command line: **-fttrap=%all,no%inexact**. We have not researched this for the other UNIX flavors.

15.11.2 gdb

gdb, a GNU product, can do four general types of things to help you debug your programs:

- Start your program, and indicate anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened when your program stops.
- Modify your program, allowing you to experiment with correcting one bug and go on to find another.

You can use **gdb** to debug programs written in C or C++.

You can also debug programs written in FORTRAN, although **gdb** does not yet support entering expressions, printing values, or similar features using FORTRAN syntax. Furthermore, it may be necessary to refer to some variables with a trailing underscore.

See the document *Debugging with GDB*, document number PU0172. On the CD home page, follow the documentation link.

15.11.3 purify

purify is a commercial product that detects memory corruption and finds memory leaks in your executable programs. **purify** is currently available on FNALU for Solaris. The command to run it is **purify**.

See the man pages for information on its syntax, options and uses.