Chapter 1

About the MIPSpro Compiler System

The MIPSpro compiler system consists of a set of components that enable you to create executable programs using such languages as C, C++, and Fortran.

A new 32-bit mode, *n32*, was introduced with the IRIX 6.1 operating system. This new 32-bit mode has the following features:

- full access to all features of the hardware
- MIPSIII and MIPSIV instruction set architecture (ISA)
- improved calling convention
- 32 64-bit floating point registers
- 32 64-bit general purpose registers
- dwarf debugging format

This new 32-bit mode provides higher performance than the old 32-bit mode available in IRIX releases prior to 6.1. When you compile **–32** in IRIX 6.1, the chip executes in 64-bit mode (UX=1) and the software restricts addresses to 32-bits. For more information about n32, refer to the *MIPSpro N32 ABI Handbook*.

In addition, the compiler system:

- uses *Executable and Linking Format* (ELF) for object files. ELF is the format specified by System V Release 4 Applications Binary Interface (SVR4 ABI). Refer to "Executable and Linking Format" for additional information.
- uses shared libraries, called *Dynamic Shared Objects* (DSOs). DSOs are loaded at run time instead of at linking time, by the run-time linker, *rld*. The code for DSOs is not included in executable files; thus, executables built with DSOs are smaller than those built with non-shared libraries, and multiple programs can use the same DSO at the same time. For more information, see Chapter 3, "Using Dynamic Shared Objects."
- creates *Position-Independent Code*, (PIC) by default, to support dynamic linking. See "Position-Independent Code,"for additional information.

Table 1-1 summarizes the compiler system components and the task each performs.

Table 1-1 Compiler System Functional Components

Tool	Task	Examples
Text editor	Write and edit programs	*vi*, *jot*, *emacs*
Compiler driver	Compile, link, and load programs	*cc*, *CC*, *f77*, *f90*, *pc*, *as*
Object file analyzer	Analyze object files	*dis*, *dwarfdump*, *elfump*, *file*, *nm*, *size*
Profiler	Analyze program performance	*prof*, *pixie*
Procedure rearranger	Minimize paging/maximize instruction cache hit rate	cord
Archiver	Produce object-file libraries	ar
Linker	Link object files	ld
Runtime linker	Link Dynamic Shared Objects at runtime	rld
Debugger	Debug programs	dbx

A single program called a compiler driver (such as *cc*, *CC*, or *f77*) invokes the following major components of the compiler system (refer to Figure 1-1).

- Macro preprocessor (*cpp*)
- Parallel analyzer (*pca, mpc*, *fef77p*, *fef90p*)
- Scalar optimizer (*copt*)
- Compiler front end (*fec, fecc*, *fef77*, *fef77p*, *fef90*, *fef90p*)
- Compiler back end
- Linker (*ld*)

You can invoke a compiler driver with various options (described later in this chapter) and with one or more source files as arguments. All specified source files are automatically sent to the macro preprocessor.Although the macro preprocessor was originally designed for C programs, it is now run by default as part of most compilations. To prevent running the preprocessor, use the **–nocpp** option on the driver command line. For C and C++ compilations, preprocessing and front-end compilation is done by *fec* and *fecc*. For all other compilations, preprocessing is done by invocation of a separate executable named *cpp*.

If available, the parallel analyzers *pca*, *fef77p*, or *fef90p* produce parallelized source code from standard source code. *mpc* takes the output from *pca* and produces parallel C code (part of Power C). The result takes advantage of multiple CPUs (when present) to achieve higher computation rates. *pca* and *mpc* are part of Power C. Power Fortran automatically uses the parallel Fortran compiler, *fef77p* or *fef90p*, to produce parallel code. For more information about these packages and how to obtain them, contact your dealer/sales representative. The compiler front ends (*fec*, *fecc*, *fef77*, *fef90*) translate the source code into an intermediate tree representation. The compiler back end (*be*) translates the intermediate code into object code. The language compilers share the same back end (*be*), which combines optimization and code generation in one phase. (For more information about optimization, see Chapter 5, "Optimizing Program Performance.")

The linker *ld* combines several object files into one, performs relocation, and resolves external symbols. The driver automatically runs *ld* unless you specify the **–c** option to skip the linking step.

When you compile or link programs, by default, the compiler searches */usr/lib*, */lib*, and */usr/local/lib*. Certain default libraries are automatically linked. Drivers and their respective libraries are listed in Table 1-2.

Table 1-2 Compilers and Default Libraries

Compiler	Default Libraries
cc	libc.so
CC	*libC.so*, *libc.so*
f77, *f90*	*libftn.so*, *libftn90.so*, *libc.so*, *libm.so*

To see the various utilities a program passes through during compilation, invoke the appropriate driver with the **–show** option.

Figure 1-1 shows compilation flow from source file to executable file (*a.out*).

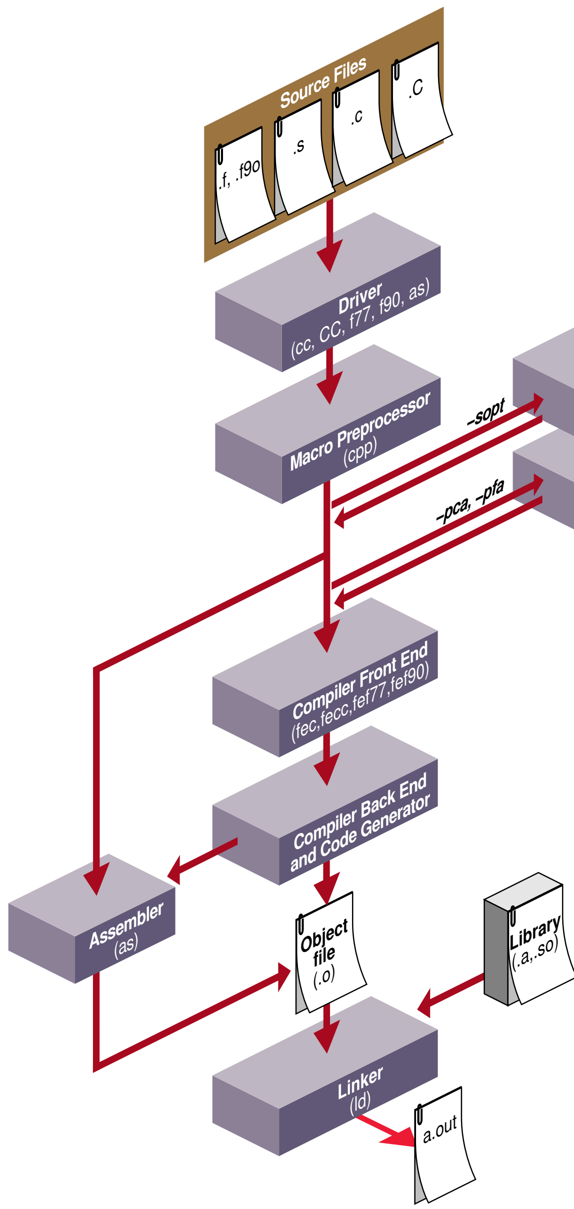

Figure 1-1 Compiler System Flowchart