INTROL-C COMPILER

REFERENCE MANUAL

Introl Corp.
647 W. Virginia St.
Milwaukee, WI 53204 USA

tel. (414) 276-2937

Table of Contents

Introl-C Compiler Reference Manual

INTRODUCTION

Introl-C provides a set of programs that have been designed to facilitate the development of high-efficiency software, in C, for microprocessor-based systems. It allows the programmer to take advantage of all the convenience, power, and structure of the C programming language, while producing executable programs whose compact size and fast speed of execution rivals that of programs written in assembly language.

The Introl-C software package includes a C Compiler, Relocating Assembler, Linker, Loader, Library Manager, and Standard Library.

This Compiler Reference Manual describes the operation, use, and features of the C Compiler and Relocating Assembler.

The operation and features of the Linker, Loader, and Library Manager are described in the Linker Reference Manual.

The Standard Library Manual furnishes a detailed description of the functions contained in the Standard Library.

Nowhere in any of these manuals do we profess to teach the C programming language. It is assumed the user has access to the definitive text, "The C Programming Language", Kernighan & Ritchie (Prentice-Hall), or one of the several available C language tutorials, for questions pertaining to the particulars of the C language itself. The set of Introl-C Users Manuals are intended only to describe Introl's implementation of the language.

GETTING STARTED

This section provides a brief overview Of the general procedures for using Introl-C and is intended to help the user get off to a "fast start" in running the Compiler and its related programs. For more detailed operating information the reader is referred to subsequent sections in this manual, as well as the other related user manuals that may have been furnished with the Introl-C package. The following comments assume that Introl-C has already been installed on the user's system. (Refer to the Installation Instructions accompanying the Introl-C distribution diskette for applicable installation procedures.)

GENERAL

Introl-C is designed to enable the user to create an executable output file from a C source file with a minimum of effort. Normally it is only necessary for the user to enter a compilation/assembly command line, and then enter a link/load command line.

In the simplest case, and assuming the C source program resides in a single file called "sieve.c", for example, all that is necessary is to enter the compiler command line:

```
icc sieve
```

and then enter the linker command line:

```
ilink sieve
```

The compiler command line entry will initiate execution of the Compiler, which first compiles the file "sieve.c" to produce an intermediate (and normally temporary) assembly language file, and then automatically calls the Assembler, which assembles the Compiler's assembly language output into a relocatable module named "sieve.R" as the result. The linker command line, in turn, will call the Linker, causing it to first link the relocatable file "sieve.R" with any referenced functions from the Standard Library, and then automatically execute the Loader, which loads the linked output into an executable output file as the final result. The executable output file will have the filename "sieve", possibly with a filename extension appended, depending upon which specific Introl Loader is being used (refer to Loader Appendices in the Linker Reference Manual for details). When the Loader finishes, three compilation-related files will typically exist: the original C source file "sieve.c", the compiled and assembled relocatable module "sieve.R", and the linked and loaded executable output file.

COMPILER COMMAND LINE

The compiler command line causes a C source file to be both compiled and assembled to produce a relocatable module as the result.

The general form of the compiler command line is:

```
icc <filename> {<option>)
```

where <filename> is the name of the C source file which is to be compiled and (<option>) represents zero or more option specifiers for controlling the compilation and assembly processes. The input filename is expected to have a filename extension; if none is specified, the Compiler will assume the source file name has the extension ".c". Unless the user explicitly assigns some other name to the output file, the relocatable file produced after the Assembler pass finishes will default to having the same name as the C source input file, except with the filename extension ".R".

Compiler-related as well as Assembler-related options may be specified on the compiler command line. Each of the available options are described in detail in the Compiler Section of this manual. Some of these option specifiers, and their general function, are indicated below.

Compiler-specific option specifiers include:

-a[t|d|b|s]=<loc>
: Causes data of type "Text" or "Data" or "Bss" or "String", respectively, to be placed under the location counter indicated by the <loc> number.

-b=<directory>
: Identifies <directory> as being the place to find current and subsequent passes of the Compiler.

-C
: Overrides default condition with respect to generation of position independent code.

-d
: Overrides default condition with respect to generation of position independent data.

-g<C>
: Forces use of alternate "<c>" version pre-processor pass.

-i=<directory>
: Identifies <directory> as a place to search for #include files.

-k
: Causes console to display the name of each compilation pass as it is being executed.

-m<name>(=<string>)
: Defines <name> in preprocessor, with value <string> optionally assigned to <name>.

-r
: Retains the intermediate assembly language output file produced by the Compiler.

-S
: Causes "nested comments" to be disallowed.

-t=<directory>
: Places temporary files produced by this and subsequent passes of the Compiler in "directory" location.

-y[=<n>]
: Strips all identifiers to a maximum length of <n> characters.

-z
: Interprets "\n" (ie newline) characters as being carriage returns.

Assembler-specific options include:

-o=<filename>
: Assigns the name <filename> to Assembler's output object file.

-q=<class>
: Sets class specifier of Assembler's output module to the numeric value indicated by <class>.

-U
: Forces all undefined symbols to default to imported symbols.

-X
: Prevents an object file from being produced.

LINKER COMMAND LINE

Unless the user explicitly opts to inhibit loading, the linker command line will cause an input module to be both linked and loaded to produce an executable output file as the end result.

The general form of the linker command line is:

```
ilink <file> {<options>} <file> {<options>} ...
```

where each <file> entered represents the name of a file to be linked and {<options>} represents zero or more option specifiers for controlling the linking and loading processes. Each input file is expected to have a filename extension; if none is specified, the Linker will assume the input filename extension to be ".R". Normally, the name of the executable output file will be the same as the module which contains the "primary function name", but with a filename extension determined by the particular Loader being used (refer to the Linker Reference Manual for further discussion).

Each file that is input to the Linker is expected to be a relocatable module. The Linker will NOT complain about producing an output module which contains unresolved references; however, attempts to subsequently load such a module will not be successful.

Both Linker and Loader options may normally be specified on the link command line. These options are discussed in detail in the Linker Reference Manual. Following are some of the link-time options that are available:

-b
    Do not search Standard Library, "libc.R".

-c=<file>
    Find additional options and/or filenames in command, file named <file>.

-d[<c>]
    Use optional "<c>ld" Loader instead of the "standard" Loader.

-e=<symbol>
    Set entry point to <symbol>.

-f<string>
    Find additional library named "lib<string>.R" in the standard place for libraries.

-f=<string>
    Find additional library named "<string>.R" in the standard place for libraries.

-l[s][x][u][=<file>]
    Produce a linker listing with specified content.

-m=<symbol>
    Set the primary function name to be <symbol>.

-n
    Do not automatically call Loader.

-o=<file>
    Assign the name <file> to Linker's output file.

-P[<c>]
    Pipe Linker's output to Loader (if applicable for host operating system).

-s
    Strip output file of all non-entry defined symbols.

-t=<classlist>
    Link using <classlist> classes of module, if they are available.

-W
    make executable file no matter what! (ie even if unresolved references exist).

FILENAME CONVENTIONS

In general, the full legal filenames of any files which are input to, or output, by, the Compiler, Assembler, Linker, and Loader are always of the form:

<name><extension>

where <name> is the nominal "generic name" of the original source file involved and <extension> is a filename extension, typically consisting of a period (.) followed by one or more trailing characters. When an input file is being specified on a command line, however, it is normally sufficient to specify just the <name> portion of the filename; the Introl-C program being called, whether it be the Compiler, the Assembler, the Linker, or the Loader, will automatically select the named file having an appropriate extension (if such file exists) as described below.

Whereas the generic name associated with a given file serves to generally identify that file as being derived from or related to some C source program or function, the filename extension indicates the specific nature of the contents of that particular file; ie whether it is a file that contains the C source text itself, or a file that contains the assembly language version of the source program, or a file that contains a relocatable module version, or a file that contains executable output, and so on.

Because of this convention of using a filename extension to identify the specific nature of a file's contents, the Compiler, the Assembler, the Linker, and the Loader are all designed to automatically append a filename extension to the output files they produce. In each case the "generic name" of the output file that each of these component Introl-C programs produces usually remains the same as that of the input file, but the extension appended to the output is unique to the particular Introl-C compilation program that generated the file. For example the Compiler normally appends an extension of the form ".M<xx>" to the assembly language files it produces, where the <xx> represents a 2-digit number as described later in this section; the Assembler appends the extension ".R" to the relocatable output files it produces; and the Linker appends the extension ".RL" to the linked (but unloaded) relocatable output files it produces. In the case of the Loader, the specific filename extension (if any) appended to the output is determined by which of the several Introl Loaders is being used to generate the executable output file.

Similarly, the Compiler, the Assembler, the Linker, and the Loader each expect their respective inputs to normally have a specific filename extension (ie usually the extension that is appropriate to the "type" of file format each of these programs expects to process). In the case of the Compiler, input files are expected to have the filename extension ".c", which is the extension normally associated with files containing C source text. Input files to the Assembler are normally expected to have an extension of the form

".M<XX>" (where <xx> represents a 2-digit number assigned by the Compiler), which is the extension normally appended to assembly language files that have been produced by the included compiler. The Linker expects its inputs to have the extension ".R", which is the extension the Assembler typically appends to the relocatable modules it produces. The Loader expects its input files to have the extension ".RL", which is the extension the Linker normally appends to the relocatable and linked output files it produces.

Thus, unless some other filename extensions are explicity defined for use on a command line, Introl-C will default to using input files, and producing output files, having filename extensions as follows:

Introl-C	Default Filename	Extension
Program	Input Files	Output File
Compiler	".c"	"M<xx>"
Assembler	".M<xx>"	".R"
Linker	".R"	".RL"
Loader	".RL"	(varies with Loader type)

*Note: The "xx>" designator in the ".M<xx>" extension represents a 2-digit number unique to the specific Introl-C compiler package that is being used. For those Introl-C compiler packages that target the 6809 processor, the specific default extension is ".M09"; for versions that target the 6801 and similar processors, the extension is ".M01"; for versions that target the 6805, the extension is ".M05"; for versions that target the 68000, the extension is ".M68"; for versions that target the NS16000, the extension is "M16"; for versions that target the 8086, the extension is ".M86".

Also, as indicated in the above table, the output filename extension that is assigned to the executable output file will be dependent upon which of the several available Introl Loaders is being used. The reader is referred to the Loader Appendices of the Linker Reference Manual for further information pertaining to Loader output filenames.

ASSEMBLER COMMAND LINE

Normally the Assembler is invoked by the Compiler automatically as part of any compilation/assembly process. However, the Assembler may also be called independently by the user for assembling user-written assembly language programs.

The general form of the assembler command line is:

```
r<xx> <filename> {<option>}
```

where "r<xx>" represents the Introl filename of the applicable Assembler furnished with the Introl-C package, <filename> is the name of the assembly language file which is to be assembled, and {<option>) represents zero or more assembler option specifiers.

The "<xx>" in the "r<xx>" filename of the Assembler is a 2-digit number unique to the specific Introl-C package being used. The Introl-C package that targets the 6809 processor has the specific Assembler filename "rO9"; the version that targets the 6801 and similar processors has the Assembler filename "r0l"; the version that targets the 6805 has the Assembler.filename "r05"; the version that targets the 68000 has the Assembler filename "r68"; the version that targets the NS16000 has the Assembler filename "rl6"; the version that targets the 8086 has the Assembler filename "r86".

The assembly language input file is expected to have a filename extension; if none is explicitly specified, the input filename extension will default to the ".M<xx>" extension that the included Compiler normally appends to its own output files. (ie ".M09", ".M05", etc, as applicable). The relocatable output file created by the Assembler will nominally have the same name as the input file, but with the filename extension ".R".

LOADER COMMAND LINE

Normally the Loader is called automatically by the Linker as a result of a linker command line call. However, the Loader may also be executed independently by the user via a loader command line of the general form:

```
<c>ld <filename> {<option>}
```

where the <c> represents the first letter of the Introl filename of the Loader which is to be called (several types of compatible Loaders are optionally available and potentially usable with Introl-C), <filename> is the filename of the relocatable file which is to be loaded, and (option) represents zero or more option specifiers. The relocatable input module is normally expected to contain no unresolved references. The input file is expected to have a filename extension; if none is explicitly specified, a ".RL" filename extension is assumed. The user is referred to the Loader Appendices of the Linker Reference Manual to determine the "<c>ld" name(s) of the specific Loader(s) that may be legally accessed, the applicable options available for each such Loader, and the unique filename extension (if any) assigned to the executable output file produced by each Loader type.

THEORY OF OPERATION

The creation of an executable file from a C source file can be considered to occur in four distinct phases: a compilation phase, followed by an assembly phase, followed by a linking phase, followed by a loading phase. Under Introl-C, however, the assembly phase is always initiated automatically when the compilation phase terminates, and the loading phase is initiated automatically when the linking phase terminates. Thus, it will normally appear to the Introl-C user as though only two phases are actually involved: a compilation/assembly phase (which is initiated via a single compiler command line call), and a linking/loading phase (which is initiated via a single linker command line call).

COMPILATION PHASE

The compilation phase, per se, is performed by the Compiler and translates a C source text file into an assembly language text file which is suitable for input to the Assembler.

The Compiler converts a C source file into assembly language by seqentially executing four separate compilation programs, or "passes", which are called passes "cO", "c1", "c2", and "c3", respectively. (Note: The "cO" pass is alternatively called the "icc" pass for some operating system versions of Introl-C.) Each of these passes performs a unique function in the overall compilation process and, as each pass finishes, it automatically initiates the next pass in the sequence.

The basic function of the c0 pass, also known as the "preprocessor", is to preprocess the C input text, removing comments and other white space from the C-source text and executing any preprocessor directives, ultimately transforming the original C input into a series of tokens that can be more easily manipulated and analyzed. If illegal characters appear in the C source text, or preprocessor directives have been used improperly, the c0 pass will detect these and flag them as errors. The cl pass, also called the "parser", converts the output of the cO pass into two resultant files: a triple file, which is a tree representaion of the original program, and a symbol file. The cl pass also checks the program for semantical and grammatical accuracy and is responsible for detecting and reporting any errors of this type. The function of the c2 pass, also called the "optimizer", is to optimize the triple file generated by cl to reduce the size and increase the execution speed of the final program. The c3 pass, called the "code generator", uses the optimized triple file produced by c2, together with the symbol table produced by cl, to produce an assembly language output file for the target processor. The several Compiler passes transfer information between one another via temporary files, which are normally automatically deleted once their contents are no longer needed by the Compiler.

The final result of the 4-pass compilation phase, therefore, is the creation of an assembly language text file which is suitable input for the Introl Assembler. Just before the last Compiler pass (c3) terminates, it automatically calls the Assembler.

ASSEMBLY PHASE

The function of the assembly phase is to translate the assembly language text file that is produced by the c3 pass of the Compiler into a relocatable object file which is suitable input for the Linker (or, if no linking is required, for possible input directly to the Loader). The assembly phase, performed by the Assembler program, is initiated automatically when the c3 Compiler pass finishes.

During the assembly phase, the Assembler converts the assembly language file produced by the compilation phase into a "relocatable" output file that contains a single relocatable module. The Assembler's output is "relocatable" from the standpoint that all address references made within the module are independent of the module's final absolute address location in memory. It is the function of the Loader to determine the final location of the module in memory and, thus, the absolute location of addresses. Therefore, until the Assembler's output module has been processed by the Loader, the output module generated by the Assembler is "relocatable" because the actual position of the module in memory is still subject to change.

Although the Assembler is capable of generating error messages, it should remain silent if the input file is the result of a compilation since the Compiler itself should in no case produce a syntactically incorrect assembly language file.

When the Compiler calls the Assembler, it normally specifies an option to the Assembler which causes the Compiler's assembly language output file to be deleted after the Assembler has finished using it. Thus, only the relocatable object file generated by the Assembler normally remains as the final result for the typical compilation/assembly process.

LINKING PHASE

The function of the Linker is to resolve external references in a relocatable module. It does this by joining the module to other relocatable modules which satisfy those external references. The result of the linking process is always a single resultant relocatable module which, if all external references have been satisfied, is suitable input for the Loader. Since the Linker normally calls the Loader automatically, it usually appears as if the Linker both links and loads the input to produce an executable file as the end result.

Whenever a program module references a label which is not defined in that same module, it is said to have an "external reference". All such external references must be "resolved" before the module can be loaded to produce an executable module. Although it is possible to create a program module that makes no external references, it is more common that a module will reference many labels which are not defined in its text; this is certainly the case with modules produced as a result of compiling and assembling a C source file. The Linker "resolves" such external references by first locating other modules which define the unresolved labels, and then linking these modules with the original module to produce a larger single relocatable module that includes the necessary label definitions. The Linker attempts to resolve as many external references as it possibly can, terminating when it either has resolved all the external references that are made or, alternatively, when it runs out of places to look for definitions which will satisfy any remaining unresolved references. When the Linker determines it has resolved all the references it possibly can, it will normally automatically call the Loader. The Linker will not complain if some unresolved references still exist in its linked output; however, attempts to load such modules will not be successful.

Inputs to the Linker must be relocatable modules, such as those produced by the Assembler, or as produced by the Linker itself (ie modules previously produced by executing the Linker alone, with the Loader pass inhibited). Normally the Standard Library is always searched by the Linker in its attempt to resolve necessary references.

LOADING PHASE

During the loading phase, the Loader fixes absolutes addresses for relocated values within a relocatable module, thereby converting a relocatable module into an "executable" output file. The exact format of the "executable" output file that is produced during the loading phase is determined by which of several optionally available Introl Loaders is being used. Depending on Loader type used, the output file may be executable under the host operating systems or executable under some other target operating system, or it may be a file of load records in one of several hex formats. (See the Loader Appendices of the Linker Reference Manual for further information.)

Normally, unless optionally overridden by the user, the 'standard' Loader included in the Introl-C package is automatically called by the Linker when the Linker terminates. For resident Introl-C compilers, the "standard" Loader is one which produces an output that is executable on the host system. For Introl-C Cross-Compiler packages, the "standard" Loader is one that produces an output file of hex load records.

The Loader expects its input to be a single relocatable module which has no unresolved external references. Normally (unless optionally overridden by the user) the Loader will complain about unresolved external references in its input and loading of such modules will not be successful.

COMPILER

The function of the Compiler is to translate a C source file into an assembly language text file which is suitable input for the Introl Assembler. In normal operation the Compiler always calls the Assembler when the it finishes. Therefore, invoking the Compiler will typically result in a fully compiled, fully assembled relocatable output module being produced.

The result of a successful compilation will be the creation of a relocatable object module which will have the same file name as the original C source input file, but with the filename extension ".R". An intermediate assembly language file is produced by the Compiler which is used as the input to the Assembler. However, this intermediate assembly language file is normally automatically deleted when the Assembler finishes using it. If the user wishes to retain the Compiler's assembly language output, a Compiler option for doing so (the "-r" option) is provided. When the "-r" option is specified, the assembly language output will be saved in a file having the same name as the C source input file, but with a filename extension of the form ".M<xx>", where <xx> represents a 2-digit number as described below.

COMPILER COMMAND LINE

A complete 4-pass compilation and assembly is initiated using a compiler command line of the following form:

```
icc <filename> {<option>}
```

where <filename> is the name of the C source file which is to be compiled and {<option>} is zero or more Compiler and/or Assembler option specifiers. (Remember the Compiler automatically calls the Assembler when it finishes.) If no filename extension is specified for the input file, the filename extension ".c" is assumed.

The result of a successful compilation and assembly will be a relocatable object module, normally having the same filename as the input file, but with the filename extension ".R" (assigned by the Assembler). The "-r" option must be specified (see Compiler Options, below) if the user wishes the Compiler's assembly language output file to be retained; this assembly language file will otherwise automatically be deleted when the Assembler finishes using it. The Compiler's assembly language output file, if saved, will have the same filename as the original input file, but with a filename extension of the form ".M<xx>", where the <xx> represents a 2-digit number. For Introl-C Compilers that target the 6809 processor, this extension will be ".M09"; for Compilers that target the 6801 and similar processors, the extension will be ".M01"; for 6805 targets, ".M05"; for 68000 targets,".M68"; for NS16000 targets, ".M16"; for 8086 targets, ".M86".

It should be noted that the Compiler pre-pends an underscore ("_") at the beginning of each symbol it generates. Thus, although a keyword such as "main", for example, is not preceded by any underscore at the C programming level, it will have a pre-pended underscore whenever it appears in any output files generated by the Compiler. Accordingly, the Assembler and Linker expect all C symbols in their inputs to begin with an underscore. Because of this, when the user is writing assembly language programs for direct input to the Assembler, or explicitly defining a "program naming function" symbol or an "entry point" symbol at link time, any C language symbols or C functions that are used must similarly always begin with a leading underscore character (even though these symbols or functions, at the C program level, do not have a leading underscore in their names).

COMPILER COMMAND LINE OPTIONS

As indicated above, option specifiers for altering the operation of the Compiler, and also the Assembler, may be specified on the compiler command line. Any such option specifiers should always appear *after* the input file named on the command line. Option specifiers are indicated by a dash, "-", followed by an alphabetic character, perhaps followed by an equals sign and parameter. The alphabetic character indicates which option is desired and the parameter is dependent on the option. Option specifiers which are not pertinent to the Compiler itself are automatically passed on to the Assembler when it is subsequently called by the Compiler. The various options available for use are described below, grouped according to whether they apply specifically to the Compiler, per se, or whether they apply specifically to the Assembler pass.

Compiler-specific options include:

-a[t|d|b|s]=<loc>

where [t|d|b|s] indicates a single letter ("t" or "d" or"b" or "s") and <loc> is an unsigned number between 0 and 15. This option will force the Compiler to place generated output of a specified type under any one of 16 available location counters, which counters are numbered from zero through 15. Data type is specified by the letter entry; "t" for text; "d" for data; "b" for bss; and "s" for strings. The <loc> entry specifies the location counter number. Thus the option specification "-ad=5" will cause all initialized data to be placed under location counter 5 (rather than its default counter of 1). The default location counter for code (text) is zero (0); the default for data is location counter one (1); the default for strings is location counter two (2); and the default for uninitialized data (bss) is location counter three (3).

-b=<directory>

This option is used to specify that <directory> is the place in which this, and subsequent passes, can expect to find subsequent passes of the Compiler. This directive may be applied to any pass of the Compiler and is in force during subsequent passes.

-c

This option changes the Compiler's default condition with respect to the "position dependency" of generated code, as follows. If Introl-C is being run on a host operating system which does not permit position dependent code to be executed, the compiler will default to generating only position independent code. In such case, this option will override this default condition and force the Compiler to instead generate position dependent code. If Introl-C is instead being run on a host operating system that does permit position dependent code to be executed, the Compiler will default to generating position dependent code. In such case, this option will override this default condition and force the Compiler to instead generate position independent code. Position independent code is code in which no absolute references are permitted; all jumps are relative to the program counter and thus are not dependent on the final location of the code in memory. This option is useful primarily for users who wish to generate code for a target machine other than the host. This option is used only by the c3 (code generating) pass of the Compiler; it may, however, be specified in the initial call to the first pass of the Compiler.

-d

This option changes the Compiler's default condition with respect to the "position dependency" of generated data, as follows. If Introl-C is being used on a host operating system that does not permit programs with position dependent data to be executed, the Compiler will default to generating only position independent data. In such case, this option overrides this default condition and forces the Compiler to instead generate position dependent data. If Introl-C is instead being run on a host operating system which does permit programs with position dependent data to be executed, the Compiler will default to generating position dependent data. In such case, this option overrides this default condition and forces the Compiler to instead generate only position independent data. Position independent data is data that must be referenced through a register. The actual position of position independent data is not known until the necessary registers are set, just prior to execution of the main program. This option is useful primarily for users who wish to generate code for a target machine other than the host. Although this option is used only by the c3 (code generating) pass of the Compiler, it may be specified in the initial call to the first pass.

-g<c>

This option specifies that an optional parser pass, named "cl<c>", be used (if such optional "cl<c>" pass exists) for the compilation process in lieu of the "standard" cl parsing pass. Depending upon the specific host operating system for which it has been supplied, some versions of the Introl-C Compiler may include the "standard" cl pass program as well as one or more optional" variations of the cl pass. The "standard" cl pass supports all features of the C language described in the "Definition Of Introl-C" section of this manual. The "optional" parser(s) provided, if any, typically omit support for one or more features of the C language and are usually intended to permit the user to circumvent memory limitations that might otherwise prevent compilation of large programs under certain host operating systems. If any optional parsers have been supplied for use for your particular host configuration, such parsers will be described in the Appendices of this manual. The option, of course, should only be specified if optional "cl<c>" parser programs have, in fact, been furnished with your Compiler.

-i=<directory>

This option specifies that <directory> is the place to search for files specified via a #include preprocessor directive if the specified file cannot be found in the default locations. This option may be specified up to 9 times so that up to 9 different places may be searched when the preprocessor is looking for an include file. If the Compiler passes are being run individually, this option is legal only for the c0 pass.

-k

This option causes the name of each compilation pass (including the assembly pass) to be displayed on the console as that pass is being executed. This is useful for permitting the user to monitor the progress of a compilation sequence when Introl-C is being run under a relatively "slow" host operating system.

-m<name>{=<string>}

This option has the effect of permitting a #define preprocessor directive to be specified on the command line. The -m option "defines" the identifier given by <name> to the preprocessor and assigns the value given by the optional <string> to this identifier.

-n

This option prevents the next compilation pass from being loaded when the current pass terminates.

-r

This option specifies that the assembly language source file produced by the Compiler (which will have a filename extension of the form ".M<xx>") should be retained. This assembly language file output by the Compiler is otherwise automatically deleted when the Assembler has finished using it.

-s

This option instructs the Compiler to disallow nested comments. That is, a slash-star combination appearing within a comment will not be interpreted as the start of a nested comment when this option is specified. This option should not be confused with the "-s=<size>" option described below, which is intended to provide a completely different effect.

-s=<size>
When the c2 (optimizer) pass of the Compiler is being executed separately, this option may be used to set the maximum size of the triple buffer. The buffer size will be set to the value indicated by <size>, which must be an integer number. Normally the size of the triple buffer is not of concern to the programmer and is otherwise automatically set by the cl pass to produce an efficient buffer size. The "-s=<size>" option should be used only when the c2 pass is being independently executed; if used under any other condition, the Compiler will otherwise interpret it as being the "-s" option,, described previously, which disallows nesting of comments.

-t=<directory>
This option specifies that <directory> is the place in which this and subsequent passes of the Compiler are to place and find their temporary files.

-Y[=<n>]
This option forces the Compiler to strip all of its identifiers to a maximum length of <n> characters, where <n> is a positive integer less than or equal to 90. If this option is not used, the Compiler will default to permitting identifiers to be up to 90 characters long. The "=<n>" entry is optional and, if not used, will cause the maximum length to be automatically set at 8 characters (ie the specification "-y" will strip all identifiers to a maximum length of 8 characters, just as would occur for the specification "-y=8").

-z
This option causes all "\n" (newline) character constants to be interpreted as being carriage returns. This option is included because the definition of the "\n" character is ambigious on some operating systems. A "\n" is defined by the C language to represent both a newline and a linefeed. This works only if the operating system in use defines its newline character to be a linefeed. Unfortunately some operating systems use the carriage return to indicate a newline. Thus, from the Compiler's point of view, it is not always clear whether a linefeed or a newline is intended by the user when a \n character is encountered. This option is provided primarily for those users having trouble with the distinction when transporting source code from one type of system to another.

The following Assembler-specific options may be specified on the compiler command line:

-o=<filename>
This option allows the user to explicitly name the Assembler's output file, assigning the name indicated by <filename> to this output file. For example, the specification "-o=file" would assign the name "file.R" to the relocatable module produced by the Assembler. If the -o option is not specified, the object

file is given the same name as the input file, except with the filename extension ".R". Unless the <filename> explicitly defines some other filename extension, the extension ".R" will automatically be appended by the Assembler.

-q=<class>

This option is used to assign a numeric class specifier to the relocatable module produced by the compiler. The class specifier assigned is determined by the <class> entry, which can be any number from zero through 255. If this option is not specified, the relocatable output module produced by the Assembler will be assigned the default class number of zero ("0"). A module's class number becomes significant when multiple modules exist which have identical "filenames"; in such instances, use of a different class number for each such module permits any given module to be uniquely identifiable.

-u

This option forces all undefined symbols to default to imported symbols. When this option is not specified, any symbol which is not imported and also not defined within the file will generate an error message.

-X

This option prevents an object file from being produced.

COMPILER ERROR MESSAGES

Compiler error messages typically occur because of one of three basic types of "errors" being encountered during compilation. The most common cause of an error message is that a syntax error of some type has been detected in the C source input file. A second type of error is when the Compiler cannot, for some reason, perform its compilation; for example, if the disk becomes full while the Compiler is attempting to write out one of its many temporary files. The third type of error is one in which the Compiler fails to operate due to an internal bug. This last type of error should, of course, never occur but a realist should not be totally unprepared for such a possibility.

Program error messages have the form:

```
file: <name> error at line <line> <message>
```

where <name> is the name of the file involved, <line> is the line number in that file at which an error became apparent to the Compiler, and <message> is a note from the Compiler which indicates what the Compiler found unacceptable. Notice that the line number given is the line in which a syntax error of some type first became evident to the Compiler. This may or may not be the actual line in the file where the program first began deviating from what the programmer may have had in mind when he was writing it. There is really no way for the Compiler to guess what the "real" error in a program may be; the Compiler can only complain at the point where the program text subsequently becomes syntactically incorrect. This may be many lines after the line which contains the actual programming error. Similarly, the message which the Compiler prints out indicates what the Compiler sees the problem to be; this may or may not be the problem as the programmer sees it.

The following are some explanations of the less obvious error messages produced by the Compiler.

'while' expected
: The Compiler expected a "while" to follow a "do" but instead found something else.

arithmetic type required
: The Compiler expected an expression which evaluated to an arithmetic type, but instead found something else such as a structure or union.

bad &
: The ampersand operator was used on something which was not an lvalue.

bad break
: A break was encountered which was not in either a "do", "while", or "for" loop, or in a "switch" statement.

bad case
A case label statement was encountered which was either outside of a switch statement or was already defined.

bad cast
The Compiler couldn't force the desired cast. This happens when one attempts to cast an integer as a structure, for example.

bad continue
A continue statement was encountered which was not in either a "for", "do", or "while" loop.

bad default
A default was encountered outside of a switch statement or else more than one default was specified for a given switch statement.

cannot create output file
The Compiler was unable to create the output file. This is usually because the disk is full.

cannot open #include file
The Compiler was unable to open the specified #include file. This is often because the user does not have permission to read the file.

compiler bug
You should never see this error. It indicates an internal error in the second pass of the compiler.

declaration of parameter not in parameter list
Indicates that a variable was declared in a function header which was not part of the parameter list for that function.

expression stack overflow, aborting
The Compiler's internal stack (on which it evaluates expressions) has overflowed. This can be remedied by breaking up the offending expression into smaller expressions which can be evaluated separately.

function required
This indicates that some expression which is not of type function is being used where a function is required.

illegal #else
An #else was encountered outside of an #ifdef or #ifndef block.

illegal #undef
This usually means that there was no identifier following the #undef keyword.

illegal array reference
An attempt was made to reference an array in an illegal fashion.

illegal character

An illegal character was encountered in the input file. This is usually due to a preprocessor directive which does not begin in column 1 but may also be caused by a missing open quote or open comment. Most control characters are considered illegal.

illegal return type

The return type of a function was not of simple type. No structures or unions may be returned as function values (although pointers to them may be returned).

label used but not defined in function

A label was used on a goto but was never defined. Labels are always local to the function in which they are defined.

lvalue required

This means that the Compiler expected an expression which could be used to represent a changeable value but did not find one. An lvalue is a value which represents a changeable value. For example if the variable XX is defined as an integer then it may be used (almost) anywhere an integer constant can be used. But it may also be used in places where it is illegal to use a constant, like on the left hand side of an assignment operator. Thus XX is an lvalue whereas a constant is not.

missing "'" or character constant too long

This indicates that more than one character was found in a quote constant. Either the terminating "'" is missing or there is more than one character between the starting "'" and the terminating "'". Cnntrol characters which begin with a backslash are considered to be a single character.

missing member name

A reference to a member name was made which was not declared to be a member of the original structure.

multiple symbol definition

Indicates that the symbol following the dash has been defined more than once.

no matching #if for #endif

An #endif was encountered but no #ifdef or #ifndef preceded it.

pointer type required

This indicates that an operation was attemoted on an expression which should be (but is not) of pointer type.

preprocessor bug #l

You should never see this one. It indicates that there is an internal error in the first pass of the compiler.

string improperly terminated: unexpected EOF

This usually means a missing close quote.

string too long, truncated at right

This indicates that a string exceeded the maximum string constant length (the current limit is 256 characters, including the terminating NULL).

struct/union tag used but not defined in block

A structure or union tag was used but not defined in the current program file, function, or block.

structure/union size unknown

This message is generated when the size of a structure or union is required (as in the sizeof operator) but is not known because the struct or union definition has not yet been encountered.

too many #define parameters

Too many parameters in a #define directive. The current limit is approximately 25.

too many nested #ifs

Too many nested #ifdef or #ifndef directives. This includes those due to #include files. The current limit is approximately 15.

unbalanced comment

This indicates that the End Of File was encountered before a comment was completed. Remember: Introl-C allows nesting of comments. Each /* must have its own */ to terminate it.

undeclared identifier, assuming auto int

An identifier was encountered which has not been defined. The Compiler will assume it was declared as an automatic integer. Notice that this assumption may cause the Compiler to generate additional error messages if the identifier is used in a fashion which is not permitted for an auto int.

unexpected end of file, unbalanced #if, #ifdef, or #ifndef

The End Of File was encountered before an #ifdef or #ifndef was completed by an #endif directive.

unexpected end of file

The End of File was encountered while the Compiler was still trying to complete some construct. For example, if the Compiler has not yet encountered the closing brace of a function definition and encounters the EOF, it will print this message.

unmatched paren or quote in macro call ... end of file

The End Of File was encountered while the Compiler was searching for an expected close quote or a right paren.

unrecognizable preprocessor directive

This indicates that a # in column 1 was followed by an unknown directive. Check the spelling of the directive.

warning - undefined operator on pointer type

This indicates that an operation was attempted involving a pointer which is not permitted on operands of type pointer.

warning - expression with no effect, ignored

This indicates that the ComDiler has found an expression with no effect. That is, no variable is updated as a result of the expression. No code is generated for the expression.

warning - union or struct as function parameter, '&' added

This indicates that an attempt was made to pass an expression of type struct or union as a function parameter. Currently this is disallowed by the Compiler. The Compiler will insert an ampersand so that a pointer to the structure will be passed instead.

ASSEMBLER

The Assembler furnished with Introl-C is a relocating assembler designed to translate an assembly language text file, as produced by the Introl Compiler, into a relocatable object file. This object file may then be linked, if need be, to other relocatable object files and loaded to produce a file which is in executable format.

In normal usage, the Compiler always automatically calls the Assembler when the Compiler, per se, finishes. The Assembler, in turn, then assembles the output generated by the Compiler to produce a relocatable object module as the final result of a compilation. The relocatable module that is produced by the Assembler will typically have the same filename as the original input, file, but with the filename extension ".R" appended.

When the Compiler automatically calls the Assembler, the Compiler passes 3 Assembler option specifiers to the Assembler; specifically, the "-n", the "-s", and the "-z" Assembler options are passed. The "-n" and "-s" option specifiers prevent the Assembler from generating any type of assembly output listing and symbol table listing, respectively; the "-z" specifier causes the Assembler to delete its assembly language input file (ie the Compiler's output file) when it has finished using it. Although the effect of the Compiler-supplied "-z" specifier to the Assembler can be overridden via a compiler command line option (ie with the '-r" Compiler option, which forces the Compiler's output file to be retained), there is no provision made to similarly override the automatically supplied "-n" and "-s" Assembler options. All this means is that the Assembler's output listing and symbol table listing will never be available as the result of a "conventional" compilation/assembly sequence. The Assembler's output listing and symbol table are readily available to the user, however, although a 2-step process is involved: (1) first, compiling/assembling the program with the "-r" specified on the compiler command line to "save" the ".M<xx>" assembly language file produced by the Compiler, and (2) then invoking the Assembler independently to separately assemble this ".M<xx>" file, thereby generating the desired output listing and symbol table as a result. As noted in the Compiler section of this manual, all symbols appearing in any output generated by the Compiler will will be pre-pended with an underscore character, which is automatically added to all symbols by the Compiler.

As inferred by the preceeding comments, although the Assembler is nominally supplied for use by the Compiler proper, it is also possible for the user to independently call the Assembler for assembling assembly language programs directly - either assembly language files which have been previously produced by the Compiler, or assembly language programs that may have been written by the user. The ability to independently use the Assembler in this way is very useful, for example, when the user wishes to include an assembly language routine as a part of a larger overall C program, or to produce a separate assembly language program. The remainder of this Assembler Section is concerned with using the Assembler independent of the compiler for these types of purposes.

ASSEMBLER COMMAND LINE

The Assembler may be called independently by entering a line of the form:

```
r<xx> <file> {<options>}
```

where r<xx> represents the Introl filename of the Relocating Assembler, <file> is the name of the assembly language source file, and {<options>) represents zero or more Assembler option specifiers. The Assembler's assembly language input file is expected to have a filename extension; if none is explicitly specified, a filename extension of the form ".M<xx>" is assumed. The output file produced by the Assembler will be a relocatable module, normally having the same name as the input file, but with the filename extension ".R".

The "<xx>" as used in both the "r<xx>" and the ".M<xx>" designations mentioned above, represents a 2-digit number unique to the particular Introl-C compiler package being used. For those Introl-C packages that target the 6809 processor, the "<xx>" represents the digits "09"; for versions that target the 6801 and similar processors, "<xx>" represents the digits "01"; for versions targeting the 6805, "<xx>" represents "05"; for versions that target the 68000, "<xx>" represents the digits "68"; for versions that target the NS16000, "<xx>" represents "16"; for versions that target the 8086, "<xx>" represents "86". Therefore, if the Introl-C package happens to target the 6809, for example, the appropriate filename for the Relocating Assembler would be "r09", and the default extension assumed for the Assemblerls'input files would be ".M09".

ASSEMBLER OPTIONS

Assembler options are listed and described below. Some of these options may be legally specified on the compiler call line when the Assembler is being called automatically as the result of a compilation. However, most of the Assembler options are legal, or will have meaning, only when the Assembler is being called independently by the user.

-a

The "-a" option forces all symbols except those that begin with a question mark, "?", to be placed in the object file. Usually only the externals and undefined symbols are included in the object file. This Assembler option may not be legally used on a compiler command line since it conflicts with the already existing (and totally different) "-a" option provided for the Compiler proper.

-c

This option causes the output listing produced by the Assembler to be sent to the console. This Assembler option may not be legally used on a compiler command line since it conflicts with a preexisting (and totally different) "-c" Compiler option.

-i

This option forces listing of all included files. Normally, included files are not part of the output listing. This option may not be legally used on a compiler command line since it conflicts with a preexisting (and totally different) "-i" Compiler option.

-j

This option forces all symbols which begin with a question mark, "?", to be listed in the symbol table. Unless this option is used, symbols which begin with a question mark are not listed as part of the symbol table listing. The Introl-C Compiler uses such labels as targets of short jumps. They are not normally listed because they are not generally of interest to the programmer. This option will have no effect if used on a compiler command line inasmuch as a symbol table is never generated as a result of a compiler command line call. A symbol table may only be produced it the Assembler is invoked independently to assemble an assembly language file.

-l=<filename>

This option specifies that <filename> is the name of the file in which the Assembler's output listing is to be placed. This causes the listing to be placed in the named file. This option has no effect if used on a compiler command line since an output listing cannot be produced as a result of a compiler command line call. An Assembler output listing can be produced only if the user invokes the Assembler independently to assemble an assembly language file.

-n

This option prevents an assembly output listing from being produced. This is one of the three Assembler options automatically passed to the Assembler when it is called by the Compiler. This option may not be legally specified on a compiler command line since it conflicts with a preexisting (and totally different) "-n" Compiler option.

-o=<filename>

This option allows the user to explicitly name the output file, and assigns the name <filename> to it. If this option is not specified, the object file will otherwise be given the same name as the input file, but with the filename extension ".R". If the <filename> that is assigned via this option does not include a filename extension, the default filename extension ".R" will be appended by the Assembler. This option may be legally specified on a compiler command line.

-q=<class>

This option assigns the class number indicated by <class> to the output object file generated by the Assembler. The <class> entry may be any number from zero ("0") to 256. If this option is not used, the module's class specifier will default to being class zero (ie "0"). A module's class number is a file identification attribute and is usually of importance only if identical filenames are assigned to several separate modules by the user; in such case, the class number attribute allows any specific module to be unambiguously distinguished from all other identically named modules. This option may be legally used on a compiler command line.

-s

This option suppresses the listing of the symbol table. This option is one of the three Assembler options automatically passed to the Assembler when it is called by the Compiler. This option may not be legally specified on a compiler command line since it conflicts with a preexisting (and totally different) "-s" Compiler option.

-u

This option forces all undefined symbols to default to imported symbols. Without this option any symbol which is not imported and also not defined in the file will generate an error message.

-x

This option prevents a relocatable object file from being produced. This option may be legally specified on a compiler command line.

-z

This option deletes the Assembler's input file when the Assembler has finished using it. This is one of the three Assembler options passed to the Assembler when it is automatically called by the Compiler: it is the option responsible for causing the the Compiler's output file to be normally deleted when the Assembler has finished using it. The effect of the "-z" specifier that is normally supplied by the Compiler in such case can be nullified by specifying the "-r" Compiler option on the compiler command line, as was mentioned. earlier. The '-z" Assembler option may not be legally specified on a compiler command line since it conflicts with a preexisting (and totally different) "-z" Compiler option.

DEFINITION OF LEGAL INPUT

This section describes the legal input to the Introl Relocating Assembler.

INPUT FILE SPECIFICATION

The input file expected by the Assembler is an ASCII text file which contains assembler text. If the input file has been generated by the Compiler it will already have an acpropriate ".M<xx>" extension, as discussed previously. If the file named on the assembler call line has no extension specified, the Assembler will attach the appropriate ".M<xx>" extension before it attempts to locate the file. A file's extension is assumed to consist of a period and any trailing characters.

INPUT LINE

Each line input to the Assembler is assumed to have the form:

```
[<label>] [<opfield> [<operand>{,<operand>}]] [<comment>]

  or

*<comment>
```

```
where  <label>    represents a symbol,
       <opfield>  represents an opcode or pseudo-op,
       <operand>  represents an expression,
  and  <comment>  represents any string of characters.
```

Those items enclosed in square brackets "[" and "]" are optional, while an item enclosed in curly brackets, "{" and "}", may be repeated zero or more times. Thus an input line may consist of an optional label, followed by at least one space, followed by an optional opfield, followed by at least one space, followed by zero or more operands separated by commas, optionally followed by at least one space and a comment. If a label is specified, it must begin in column one. It is also legal to indicate an entire line as being a comment by placing a star, "*", in column one. If no label is specified, column one must be a blank or a star. An example of a legal input line:

```
loop   jmp    loop          This is VERY tight loop

  or

* This whole line  is a comment
```

SYMBOLS

Symbols are made up of letters (a..z, A..Z), digits (0..9), the question mark (?), the dollar sign (s), the underscore (_) and the period (.). Symbols must begin with either a letter or a period or an underscore or a question mark and may be any length. In the special case of symbols that reference C functions, such symbols must ALWAYS be preceded by a leading underscore character (ie, just as the Compiler pre-pends an underscore to all symbols it generates). The first one hundred characters of a symbol are retained by the Assembler. Case is not ignored when the Assembler compares two symbols: "abc" is NOT equal to "ABC" is NOT equal to "AbC'.

Valid Symbols:

```
      .abc
      abc09
      .9
      Very.long.symbol.only.the.first.100.characters.count
      ..PIA10.
```

Although one hundred characters are significant to the Assembler, when the symbol table is output, only the first sixteen characters of the symbol are printed so that the printout will look better.

OPCODES

In general, the opcodes recognized by the Assembler are the standard opcodes, recognized by the microprocessor manufacturer's assemblers. All opcodes can be placed anywhere on the source line after the statement label, or at least one space or tab from the beginning of the source line if no label is present. Opcodes may be in either upper or lower case.

PSEUDO-OPS

Pseudo-ops are a set of mnemonics which represent commands to the Assember rather than instructions to be coded. The legal pseudo-ops are described below in the section on assembler directives.

EXPRESSIONS

The Assembler accepts assembly type expressions that are arbitrarily complex. Several operators are allowed in assembly time expressions (alternate forms listed on the same line are identical in function):

-	unary minus (two's complement)
~	not (one's complement)
*	multiplication
/	division
%	mod (remainder)
+	addition
-	subtraction
<<	shift left
>>	shift right
&	bitwise and
^	bitwise exclusive or
|	bitwise inclusive or
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
==	equal to
!=	not equal to

Operator precedence of the above operators is, from highest to lowest (alternate forms have the same precedence as regular forms):

```
        -     ~
        *     /     %
        +     -
        >>    <<
        >     <     <=    >=
        ==    !=
        &
        ^
        |
```

Parentheses are allowed in expressions to change the precedence of an expression.

Assembly time expressions can be used in the operand of any assembler opcode or directive. Symbols and constant values can be used interchangeably in an expression. All results of expressions at assembly time are 32 bit, truncated integers. Constant values are defined as a numeric digit (0..9), followed by zero or more numeric digits or the letters A..F, followed by a radix indicator.

n<radix>

where n is 0..9,A..F (must be a valid digit in the given radix), preceded by a numeric digit, and <radix> is

H	hexadecimal
O,Q	octal
B	binary
D or nothing	decimal

An alternate way of specifying constants is by preceding the constant by the alternate radix indicator followed by one or more valid digits in the given radix.

```
           <altrad>n
```

where <altrad> is

$	hexadecimal
@	octal
#	binary
& or nothing	decimal

and n is 0..9,A..F (must be a valid digit in the given radix). No preceding numeric digit is required.

Constants may also be ASCII character constants, either one or two characters long:

```
           '<ch>     is a one character constant
           "<chch>   is a two character constant
```

The Assembler also recognizes a special constant that represents the assembly time location counter: "$" or "*". When "$" or "*" is used in an expression, the value taken is the location counter at the instant of assembly of the line containing the "$" or "*".

Examples of Constants:

```
01010101B
17q
$10
17777o
"AB
567H
%0110101
0ffffh
'@
13
7FFH
$
*
```

Examples of valid expressions:

```
(start-end)>2 start minus end shifted right by two

abc*5         five times the value of abc

'a!80h        ascii value of 'a' 0Red with 80 hex

$+4           value of the location counter plus four

*-3           value of the location counter minus three

$FFFF<<(3-LABEL)+*     ?????
```

ADDRESSING MODES

All addressing modes of the microprocessor are recognized by the Assembler.

ASSEMBLER DIRECTIVES

The following is a list of assembler directives. An assembler directive is a line which issues a command to the Assembler. All assembler directives may be in either upper or lower case.

comm - Common Area

This directive has the form:

```
<label> comm <size>
```

where <label> is any legal identifier and <size> is an absolute expression which indicates the size, in bytes, which should be reserved for the label. The comm directive has virtually the same effect as the import directive except that, if the Linker cannot find any definition to satisfy the external reference, it will reserve a location in the bss segment segment of <size> number of bytes. A label may appear in any number of comm directives.

<u>dc - Define Data Constant</u>
This directive has the form:

```
[<label>] dc[.<sizecode>l <expression>{,<expression>}
```

where <sizecode> indicates an optional letter ("b", "W", or 11110) which indicates the size of the data object (byte, word, or long). The <expression> is an absolute or relocatable expression whose value is placed in the location. Multiple locations may be defined by a single dc directive by specifying multiple expressions separated by commas. Each expression will be evaluated and the resultant values will be placed in successive locations, each of which is assumed to be the size indicated by the size code letter. If the size code letter is omitted, the size is assumed to be the size of an integer (2 bytes). In the case of the dc directive it is permitted to have an expression of the form:

```
           '<string>'
```

where <string> is one or more ASCII characters. The characters will be packed into successive bytes.

<u>ds - Define Data Storage</u>
This directive has the form:

```
(<label>] ds[.<sizecode>] <size>
```

where <sizecode> indicates an optional letter ("b","w",or "l") which indicates the size of the data object (byte, word, long). The <size> indicates the number of data objects for which space is to be reserved. The number of bytes reserved is the <size> multiplied by the size of the data object (1, 2, or 4 bytes).

<u>end - End of Assembly</u>
This directive has the form:

```
           end     [<label>]
```

where [<label>] is an optional label which, if specified, causes the output module's entry point to be set to that indicated by the label. The label should be an external label which must have been defined before the occurrence of the "end" directive. This directive is used to signal the end of input for the Assembler.

<u>equ - Equate Svmbal With A Value</u>
This directive has the form:

```
           {<label>} equ <expression> {<comment>}
```

The equ directive gives the value of the expression in the operand to the label. The label and operand are both required with an equ directive; the comment is optional. The equ directive is similar in function to the "set" directive except that a symbol defined with an equ cannot be redefined elsewhere in the program. The <expression> cannot contain external references, forward references, or undefined symbols; it may, however, be relocatable.

```
    one  equ   1         equate the value 1 to one
    five equ   one*5     equate  the value  one times  5 to five
```

<u>err - Programmer-Generated Error</u>
This directive has the form:

```
         err {<string>}
```

The err directive will cause an error message to be printed by the Assembler. The total error count will be incremented as with any other error. The err directive is normally used in conjunction with conditional assembly directives for condition checking. The assembly proceeds normally after the error has been printed. The optional {<string>} may be used to specify the nature of the error generated.

<u>export - External Symbol Definition</u>
This directive has the form:

```
    export <symbol>{,<symbol>,...,<symbol>} {<comment>}
```

The export directive is used to specify that the list of symbols is defined within the current source program, and that these symbol definitions should be passed to the Linker so other programs may reference them. If the symbols contained in the operand of this directive are not defined in the program, an error will be generated.

<u>fcb - Form Constant Byte</u>
This directive has the form:

```
{<label>} fcb <expression list> {<comment>}
```

The fcb directive allows the programmer to define a byte constant or series of byte constants. The <expression list> in the fcb operand is a sequence of one or more expressions separated by commas. The value of each expression is truncated to 8 bits and stored as a single byte in the object program. Multiple expressions are stored in successive bytes. If a field between two commas is empty, a zero value is stored for that byte. The label and comment fields are optional. An error will occur if the upper eight bits of each expression in the operand do not evaluate to all zero's or all one's.

```
table     fcb 0,1,2,3,0fh,27q,7
          fcb 0,,,,,,,,,0          ten zero bvtes
          fcb five,one,4*5,'A
```

fcc - Form Constant Character

This directive has the form:

```
{<label>} fcc <delimiter><string><delimiter> {<comment>}

         -or-

{<label>} fcc <expression>,<string> {<comment>}
```

The fcc directive converts a string of characters into a sequence of bytes containing the characters' ASCII-values. Two forms of the fcc directive are available. The first form above delimits the string to be saved by a delimiter character which can be any character except the numeric (0..9) digits. The delimiter character cannot appear in the given string. The second form of the fcc directive takes two arguments, separated by a comma. The first argument is an expression representing the length of the subsequent string. The expression argument of the fcc directive must begin with a numeric (0..9) digit. The length expression represents the exact length of the resultant string: if the given string is longer than this length, the string is truncated; if the given string is shorter than this length, the string is expanded with spaces (ASCII 20H). When the length expression is longer than the given string, there is a danger that a comment, if one is given, may be taken as part of the string. It is usually better to leave comments out of this type of fcc directive.

```
msgl    fcc   'this is a  string'            "'" is the delimiter
        fcc   /this  is another string/      "/" is the delimiter
ms92    fcc   64,this is yet another
        fcc   26,abcdefghijklmnopqrstuvwxyz
        fcc   /abcdefghijklmnopqrstuvwxyz/
```

The last two examples save exactly the same sequence of bytes in memory: the 26 lower case alphabetic characters, in order.

fdb - Form Double Byte Constant

This directive has the form:

```
   {<label>} fdb <expression  list> {<comment>}
```

The fdb directive is similar to the fcb directive above except that, whereas the fcb directive causes each expression in the list to be taken as a byte value, the fdb directive instead causes each expression to be taken as a double byte, or word, value.

```
address.table
           fdb routine.l,routine.2,routine.3
           fdb routine.4,routine.6
address.table.length    equ ($-address-table)/2

           fdb 1024*48,address.table,address.table.length
           fdb "AB,01010101B,37D
```

ident - identify module
This directive has the form:

```
          ident   <name>,<class>,<rev>
```

where <name> will be the name of the output module, <class> is an integer from "C" to "255" which specifies the class number to be given the resultant module, and <rev> is a revision number to be given the resultant module. If the class or revision numbers are left unspecified they will default to zero (0). If the module name is left unspecified it will default to the filename of the assembly language input file, minus any extension.

import - External Symbol Reference
This directive has the form:

```
          import (<loc>:]<sym>{,[<loc>:]<sym>}
```

where <loc> represents an optional location counter specification and <sym> is some symbol to be imported. The import directive is used to inform the Assembler that the named symbols are referenced by the current source program but are defined elsewhere. Each symbol in the list may be preceded by an optional absolute expression whose value must be between 0 and 15. The expression indicates the location counter the corresponding symbol is assumed to be under. The Linker will issue an error message if the symbol has been specified under a different location counter than the one listed on the import directive.

If import is not used to specify that a symbol is defined in another program, an error will be generated, and all references to the symbol in the current program will be flagged as being undefined.

lib - Load A Disk File
This directive-has the form:

```
          lib <filename>
```

The lib directive makes it possible to read a disk file as part of the assembly process. The file is used as if is were actually a part of the source code being assembled. The <filename> argument should be a valid file name for the system you are using.

```
      lib MYFILE.MO9
```

list
This directive has the form:

```
        list
```

The list directive reverses the effect of a previous nolist directive. (See the nolist directive below for a description of its function).

loc

This directive has the form:

```
 loc <counter>
```

where <counter> is an integer within the range 0 to 15. This directive indicates that all code generated until the next "loc" directive will be placed under the named location counter.

nolist

This directive has the form:

```
        nolist
```

The nolist directive prevents the code following it from being listed in the assembler output listing. The nolist directive works in conjunction with the "list" directive, decribed earlier, to bracket code which is not to appear in the output listing. A nolist is in effect until a list directive appears. The list and nolist directives may be nested; therefore, in order to nullify two successive nolist directives, the Assembler must subsequently encounter two successive list directives.

offset

This directive has the form:

```
         offset <expression> (<comment>)
```

The offset directive allows the user to generate labels whose values represent absolute offsets from some origin. This is useful in defining labels which are to be used as offsets into predefined tables.

```
        offset 0            set offset at zero
data    ds.b   2            set label "data" equal to 0
data2   ds.b   1            set label "data2" equal to 2
```

rmb - Reserve Memory Bytes

This directive, which is identical to the ds.b form of the ds directive discussed previously, is defined as follows:

```
{<label>} rmb <expression>   {<comment>}
```

The rmb directive causes the location counter to be incremented an amount specified by the expression in the operand field. This reserves an area in memory whose length, in bytes, is equal to the value of the operand expression. The memory area reserved by the rmb directive is uninitialized by the directive. The expression cannot contain external references, forward references, or undefined symbols. The label and comment fields are optional.

```
xtable    rmb       256        save 256 byte for xtable
          rmb       20         save  20 bytes  for the stack
stack
data      rmb       1024*4     save 4K for data area
buffer.length equ   132
buffer    rmb       buffer.length reserve buffer space
```

set - Set Symbol To A Value

This directive has the form:

```
<label> set <expression> {<comment>}
```

The set directive assigns the value of the expression to the label. Function of the set directive is similar to that of equ except that labels defined using set can have their values redefined in another part of the program by using another set directive. The set directive is useful for establishing temporary or re-usable counters within macros.

syn - Equate Labels

This directive has the form.

```
<symbol> syn <symbol>
```

where <symbol> is any previously defined symbol. This directive makes the first symbol synonomous with the second symbol. The new symbol has all the attributes of the original. Thus the user may redefine opcodes, register names, labels, or any other symbol.

DEFINITION OF INTROL-C

This section provides a detailed definition of the Introl-C implementation of the C programming language. It assumes the reader already has a reasonable understanding of "standard" C and is not intended to serve as a tutorial on the C language.

LEXICAL CONVENTIONS

WHITE SPACE

Blanks, tabs, newlines, and comments are considered "white space". For the most part the Compiler ignores white space, although, occasionally white space may be required to separate otherwise adjacent identifiers, keywords, and constants.

COMMENTS

The character combination slash star (/*) indicates the beginning of a comment. Comments must be terminated with a star slash combination (*/). Comments are considered white space and have the same effect as a blank. Introl-C allows comments to be nested, permitting large sections of code (which may already contain comments) to be "commented out" by simply bracketing the section with /* and */. This is not possible in "standard" C since standard C does not allow nesting of comments. Introl-C provides a Compiler option (the "-s" option) to permit the user to override this "nesting of comments" feature if the user wishes to disallow nested comments. Each slash star (/*) combination used in a comment requires that a matching */ terminator also appear in the comment. That is, the following may not do what you would think:

```
/* This comment /* doesn't end at this terminator -> */
```

Comments are removed from text before preprocessor directives are evaluated; thus preprocessor directives may also be "commented out" by bracketing them with /* and */.

IDENTIFIERS

An identifier consists of an Alphabetic letter followed by zero or more letters or digits. There is no limit on the number of characters which may be used to specify an identifier, although only the first ninety (90) characters will be considered significant. A Compiler option (the "-y[=<n>]" option) is provided to permit the user to set the maximum identifier length to values less than the normal maximum of ninety characters. The underscore, (_), counts as a letter. Upper and lower case letters are considered to be different.

KEYWORDS

The following identifiers are reserved and may not be redefined by the user.

auto	double	int	struct
break	else	long	switch
case	extern	register	typedef

char	float	return	union
continue	for	short	unsigned
default	goto	sizeof	while
do	if	static	

CONSTANTS

Integer Constants: Integer constants may be represented in several different formats. A string of digits beginning with a 0 (zero) is taken to be in octal; the digits 8 and 9, if used, are taken to have the octal values 10 and 11 respectively. If the constant begins with an 0x or 0X (zero x) it is taken to be hexadecimal and the characters A through F (either upper or lower case) may be used to represent the decimal values 10 through 15 respectively. If there is no preceding 0 or 0x or 0X, the constant is taken to be decimal. A decimal constant which is greater than the largest signed integer is taken to be a long. An octal or hexadecimal constant which is greater than the largest unsigned integer is taken to be long.

Long Constants: Long constants may be declared explicitly. A decimal, hexadecimal, or octal constant which is terminated with the letter L (either upper or lower case is permitted) is taken to be long. Long constants are implemented in 32-bit two's- complement form.

Character Constants: A character constant is any graphic or non-graphic character enclosed in single quotes; 'x' for example. The value of a character constant taken to be the numerical value used to define that character in the machine's character set (usually ASCII).

The single quote character ('), the backslash character (\) and various non-graphic characters may be represented by the following character combinations:

newline	\n
horizontal tab	\t
backspace	\b
linefeed	\l
carriage return	\r
form feed	\f
backslash	\\
single quote	\'
bit pattern	\ddd Where ddd is 1,2 or 3 octal digits which specify the character's value.

*Note: Introl-C normally interprets "\-n" (ie the newline character in C) as being a linefeed character; however, a Compiler option (the "-z" option) may be used to instead equate "\n" with being a carriage return character.

Unless a backslash is used in one of the above character combinations, the backslash will normally be ignored. Character constants are represented as a single 8-bit unsigned byte.

Floating Constants: A floating point constant consists of an integer part, a decimal point, a fractional part, and an exponential part. The integer and fractional parts each consist of a string of one or more digits. The exponential part consists of an "E" (either upper or lower case), followed by an optionally signed integer. Either the integer part or the fractional part (but not both) may be missing; either the decimal point or the exponential part (but not both) may be missing.

Strings: A string consists of a sequence of zero or more characters placed between a set of double quote marks, as in "this is a string". A string has the type Array Of Characters and thus may be used anywhere an array of characters would be appropriate. All strings are treated as uniquely distinct data objects, even when they contain identical sequences of characters. The Compiler will place a null byte (\0) at the end of each string so that functions which scan the string can determine its end by the usual means. All the conventions for representing non-graphic characters which apply to character constants apply to strings as well. To represent a double quote inside a string it is necessary to precede it with a backslash. Strings may be continued on a new line by inserting a backslash followed immediately by a carriage return. The backslash carriage return combination is not considered part of the string.

PRE-PROCESSOR DIRECTIVES

A preprocessor directive is an instruction to the preprocessor (lexical scanner) which controls the input to the Compiler proper. These directives control such things as file insertion (#include), textual substitution (#define), and conditional compilation (#ifdef). Pre-processor directives always start with a pound sign (#) and must begin in column one. The effect of these directives is the controlled alteration of the program text input to the compiler. The directives supported by Introl-C are #define, #else, #endif, #ifdef, #ifndef, #include, #undef. Their function is explained below.

#define: The #define directive allows an identifier to be equated with a string. There are two forms of the define directive. One case handles simple string substitution, in which a token-string will be substituted for any occurrence of the identifier which appears in the program text following the #define statement. The other case allows parameter substitution, so that sections of the replacement string may be specified at the place in the code where the identifier is used. The first case of the #define directive, calling for simple string substitution, has the following form:

```
#define <identifier> <string>
```

where <identifier> represents the name of the identifier and <string> is any series of characters. The <string> is optional. There must be at least one space between the word #define and the identifier. This form of the define statement causes any occurrence of the identifier which appears in the program text following the define statement to be replaced with the strings. Notice that there is no semicolon required at the end of a #define directive. The <string> is taken to be all the characters which follow the identifier on the #define line. Thus, it is incorrect to place a semicolon at the end of the line unless it is actually intended to include a semicolon in the replacement string.

The second form of the #define directive looks like this:

```
#define <identifier>(<identifier>,...,<identifier>) <string>
```

This form of the define statement (called a macro definition) has a set of parameters following the first identifier. Notice that the left parenthesis of the parameter list must immediately follow the first identifier with no intervening white space. If there is any white space following the identifier, the preprocessor will interpret the #define statement as being of the simple string substitution type described above and will treat the parameter list as if it is part of the <string>. The parameter list consists of a series of identifiers separated by commas. Each identifier in the parameter list should appear at least once in the <string>. When the defined identifier appears in the program text it may be followed by an argument list enclosed in parentheses and containing strings separated by commas. If so, these strings will be substituted for their respective parameter identifiers in the <string> of the define statement before the <string> replaces the identifier in the program text.

The #define preprocessor directive has the additional effect of "defining" an identifier for use with the #ifdef and #ifndef preprocessor directives. It is permissible to have a #define statement with no <string> parameter; this will simply "define" the identifier within the preprocessor.

<u>#else:</u> This directive modifies the effect of a previously declared, non-terminated #ifdef or #ifndef conditional compilation preprocessor directive. If the lines preceding #else were being ignored because of an #ifdef or #ifndef, the #else directive will cause the lines following the #else to be processed. Likewise if the lines preceding #else were being processed because of an #ifdef or #ifndef, the lines following the #else will be ignored. The effect of the #else directive lasts until an #endif directive is encountered. The #else directive has the following form:

```
#else
```

<u>#endif:</u> This directive terminates the the most recent previously declared #ifdef or #ifndef directive. It has the following form:

```
#endif
```

<u>#ifdef:</u> The #ifdef directive is used to denote the starting point of a section of code which is subject to conditional compilations. This

directive has the form;

```
#ifdef <identifier>
```

where <identifier> represents an identifier name. If the named identifier is currently "defined" in the preprocessor, the lines following the #ifdef directive will be processed until an #else control line is encountered or, in the absence of an #else, until the #endef directive is encountered; any lines between #else (if present) and #endef are ignored for this case. If the identifier named on the #ifdef line is NOT currently defined, then only the lines between the #else (if present) and the #endef terminator line will be processed. An identifier is taken to be "defined" if it has previously appeared as the identifier on a #define preprocessor directive line. An identifier is taken to be "undefined" if it has previously appeared on an #undef preprocessor directive line, or if it has never appeared on a #define directive line.

<u>#ifndef:</u> The #ifndef directive is similar in function to #ifdef, above, except that compilation of subsequent code is conditional upon an the identifier being currently "undefined" in the preprocessor. The #ifndef directive has the form:

```
#ifndef <identifier>
```

where <identifier> is the identifier name. If the named identifier is NOT currently defined, subsequent lines will be processed until an #else control line is encountered or, in the absence of an #else, until the #endif directive is encountered; any lines between #else, (if present) and #endef are ignored in this case. If the identifier named on the #ifndef line IS currently defined, only the lines between the #else directive (if present) and the #endef terminator line will be processed. An identifier is taken to be "undefined" if it has previously appeared as the identifier on an #undef preprocessor directive line, or if it has never appeared on a #define preprocessor directive line. An identifier is taken to be "defined" if it has previously appeared on a #define preprocessor directive line.

<u>#include:</u> The #include directive causes the file specified on the #include line to be inserted in the program text in place of the #include line. Either of the following forms are permitted:

```
#include "filename"
```

or

```
#include <filename>
```

where filename is the name of the file to be included. Notice that the Introl-C compiler allows either angle brackets or double quotes to surround the filename. Included files may themselves contain include statements; that is, #include directives may be nested, with a limit imposed only by the constraints of the operating system.

#undef: The #undef directive causes the named identifier to be "undefined". Thus any subsequent #ifdef and #ifndef directives which reference the identifier will operate as if it was never defined. It has the form

```
#undef <identifier>
```

where <identifier> is the name of the identifier that is to be undefined.

DATA CONVENTIONS

All user defined identifiers have two attributes, (1) storage class and (2) type, which are described below.

STORAGE CLASS

An identifier's storage class indicates the location, scope and lifetime of the storage associated with the identifier. There are four different storage classes: auto, extern, static, and register.

auto: Automatic variables are local to the block or function in which they are defined. They exist only while the block or function in which they were defined is executing. Their contents are discarded upon exit from the block. Variables in a function which are not explicitly defined as having a specific storage class are assumed to be automatic (ie auto) variables.

extern: External variables exist for the entire execution of the program and retain their values throughout the execution of the program. An external variable may be referenced by any function in the program file in which it was defined. Also, separately compiled program files which declare external variables of the same name refer to the same variable, thus allowing communication between separately compiled program files.

In Introl-C there is little distinction made between an external "definition" and an external "declaration". It is possible to link several files together in which an external variable has been declared but never defined; the linker will simply define the variable to fit the declarations. It is also permitted to link files in which an external variable has been defined more than once; the linker will simply treat the extra definitions as if they were declarations. The linker will issue a warning if an external variable has multiple incompatible definitions in a group of files to be linked. An external variable may be initialized only once among all the program files-to be linked together.

register: The idea behind the register storage class is that it may be desirable to have a frequently used variable stored in a high speed register. The register storage class is a hint to the compiler that it should, if possible, place this variable in a high speed register. In the case of Introl-C, the compiler makes most of these kinds of decisions on its own. Specifying a variable as being of register storage class is not guaranteed to cause the variable to be placed in a register. In fact, Introl-C register variables are identical to auto variables.

static: The scope of a variable declared with a static storage class is limited to the block, function, or file in which it was defined, much like an auto variable. Unlike an auto variable, however, the contents are not discarded when the block containing the variable terminates. That is, the contents of a static variable remain valid between invocations of the defining block or function.

typedef: The typedef storage class does not actually assign storage but is simply a mechanism for associating an identifier with a data type. It is included here because it is syntactically identical to a storage class specifier. Once an identifier has been included in a typedef declaration it may be used in place of a type specifier in subsequent type declarations.

TYPE

The second attribute that may be specified for an identifier is its type. Types may be divided into two main classes, the first being the "fundamental" class of data types and the second the "derived" class of types. The derived types comprise a conceptually infinite class of types which may be constructed from combinations of fundamental types or already defined derived types. The presently supported fundamental types are:

```
char
int
float
```

where int may be optionally preceded by one of the modifiers: short, long, or unsigned.

The derived types are as follows:

```
arrays of objects of most types
functions which return objects of various types
pointers to objects of any type
structures of objects of most types
unions of objects of most types
```

The fundamental types are discussed individually below.

char: A character variable is defined to be large enough to store any character from the machine's character set (assumed to be ASCII) as a positive number. All character variables are implemented as 8 bit bytes. The Introl-C Compiler treats character variables as unsigned quantities.

int: integers are used to represent integral quantities. Integer data objects can be declared in various sizes or as signed or unsigned by use of an optional modifier (or the lack thereof). integers come in up to three sizes: "short int", "int", and "long int". Short integers are guaranteed not to be longer than an integer. Integers are guaranteed to not to be longer than a long integer. In Introl-C short integers are 16 bit quantities and long integers are 32 bit quantities. Normal integers are whatever length is most appropriate for the machine in use. (Refer to the other Appendices of this manual for further information on integers which is specific to the target microprocessor.) All signed integers are represented in 2's complement form. Unsigned integers represent positive quantities.

float: Floating point numbers are represented in the IEEE standard floating point format. A floating point variable is allocated 32 bits of storage which is interpreted by floating point functions in the following way: the most significant bit is interpreted as the sign of the number; the next 8 bits are interpreted as a biased exponent; the remaining 23 bits are interpreted as a normalized mantissa preceded by an assumed bit which is always set to 1. Floating point numbers cover the range from approximately 8.43 times 10 to the -37th power to 3.37 times 10 to the +38th power. It is also possible for floats to take on values outside this range. Such values are used to represent positive and negative infinity (+inf, -inf), and Not-a-Number (NaN). In the case of NaN the variable will be encoded in such a way as to contain an error code and an address which indicates where and under what circumstances the NaN occurred. Various printing routines will actually print out "+inf" for positive infinity, "-inf" for negative infinity, and "NaN" for Not-a-Number. In the case of NaN, two numbers separated by commas may be printed following the NaN; the first represents an error code and the second the address which was encoded in the number. (See printf and atof in the Standard Library volume).

The derived data types are described below.

Arrays: An identifier may represent an array of any type except function. Notice that an array MAY be of type pointer to function and indeed this is usually what is meant when one refers to an "array of functions."

In expressions, array identifiers are converted to a pointer to the first member of the array. The converted identifier is, of course, not an lvalue and thus may not be modified as an actual pointer might. By definition, the expression El[E2) is identical to *((E1)+(E2)). The rules for adding a pointer to an integer state that the result is a pointer which is offset from the original pointer by a number of bytes equal to the integer multiplied by the size of the object to which the pointer points. Thus if El is an array or pointer, and E2 is an integer, then both El[E2] and *((E1)+(E2)) refer to the E2th element of El. Multi-dimensional arrays are simply implemented as arrays of arrays. That is, El[E2](E3] is identical to (E1[E2])[E3]. Multi-dimensional arrays are stored row-wise in memory (the rightmost subscript varies fastest).

functions: An identifier may represent a function which can be declared as returning any one of the fundamental types as well as a pointer to any type. A function identifier may represent two different things. If it is followed by a set of parentheses (which may contain a parameter list) it is interpreted as a function call; otherwise it is interpreted as the address of the function.

pointers: An identifier may represent a pointer to any type. A pointer to a type may be thought of as a variable which contains the address of an object of that type. That is, a pointer to integer contains the address of some variable of type integer. It is possible for a pointer to point to nothing, in which case it is said to equal NULL; this is signified by setting the pointer equal to zero. Only three mathematical operations are defined for pointers. A pointer may be added to an integer, in which case the result is a pointer which is offset from the original pointer by a number of bytes equal to the integer multiplied by the length of the object pointed to. This has the same effect as specifying the pointer with the integer as an index (see arrays above). An integer may be subtracted from a pointer, with an effect identical to adding the negated integer to the pointer. Thirdly, a pointer may be subtracted from another pointer, in which case the result is an integer representing the number of objects separating the objects being pointed at. This last operation is defined only when both pointers point to objects in the same array.

structures: An identifier may represent a structure whose elements may be of any type except "function". (See the note in "Arrays" above). A structure allows a set of variables of various types to be grouped under a single name for convenience. The only operations which can be performed on a structure are (1) to take its address (using the "&" operator), and (2) to access one of its members. Functions may not be assigned or copied as a unit nor may they be passed to or returned from functions (pointers to structures may be passed to and returned from functions, however). When referencing structure members through pointers, the construct (*<Pointer>).<member> is equivalent to <pointer>-><member>, where <pointer> is an expression which evaluates to "pointer to structure" and <member> is a member of the structure pointed to.

Introl-C provides separate name spaces for all structure and union member names, allowing identical member names to be used in different struct or union declarations with no restrictions. Thus, two different structures may each have a member with the same name. Another advantage to having all structure and union member names in separate name spaces is that the Compiler can do more extensive type-checking of structure references. To access a member of a struct or union through a pointer expression, the pointer expression must be of type pointer to the particular structure or -union in question. This type checking can be overridden if desired by using a cast to cast the pointer to the type of the structure to be accessed.

unions: An identifier may represent an object which can contain any one of several types of any type except function. (See arrays).

Introl-C provides separate name spaces for all structure and union member names, allowing identical names to be used in different struct or union declarations. Thus, two different unions may each have each have a member with the same name. The Compiler will flag as an error a reference to a union or structure member which is made with a pointer which is not of type pointer to the union or structure referenced. If it is desired to defeat this type-checking, the pointer in question may be cast as a pointer to the union or structure to be referenced. (See "structures" above).

DECLARATIONS

Declarations are the mechanism for associating an identifier with a type and storage class. There are two main types of declarations, Data Declarations and Function Definitions.

DATA DECLARATIONS

A data declaration consists of an optional storage class specifier, followed by an optional type modifier, followed by an optional type, followed by zero or more declarators (each of which may be followed by an initializer) separated by commas, followed by a semicolon, ";". The storage class specifier may be any of the following:

```
auto
extern
register
static
typedef
```

A type modifier may be any of the following:

```
long
short
unsigned
```

A type may be any of the following:

```
char
int
float
struct <identifier> {<member declarations>}
union <identifier> {<member declarations>}
<typename>
```

A declarator may be an identifier, or a declarator enclosed in parentheses, or a declarator preceded by a star, or a declarator followed by a set of empty parentheses, or a declarator followed by a set of brackets which may optionally enclose a constant expression.

All items are optional except the declarator. If the storage class is not specified and the declaration is within a function definition, then auto will be assumed; otherwise extern will be assumed. Type modifiers may appear only for a type of int, or when the type is left unspecified. If the type modifier is not specified, int will be assumed.

The typedef storage class specifier does not reserve storage but is used to associate an identifier with a data type. It is included here because, from a syntactical point of view, it is a storage class specifier.

For structure and union types either the <identifier> or the (<member declarations>) part may be omitted (but not both). That is, a structure or union type consists of the following: the keyword "struct" or "union", followed by an optional identifier, optionally followed by a set of braces which enclose a list of member declarations. A member declaration consists of an optional type specifier followed by zero or more declarators where declarators are as defined above. The <identifier> part may appear without the {<member declarations>) part, provided that the same identifier has previously appeared in a structure definition which included the (<member declaration>) part.

The type may be a <typename>, where <typename> was a previously declared identifier in a declarator which appeared in a declaration having a storage class of "typedef".

INITIALIZERS

As mentioned above it is possible for a declarator to be followed by an initializer. The initializer is a vehicle by which the programmer may specify the initial value of a variable. For external and static variables the value is set once, logically, at compile time. For automatic variables the value is assigned to the variable on each entry to the function (ie at run time).

The syntax for the most general use of initializers, as applied to external or static variables, is as follows: an equal sign, followed by an initializer-list. The initializer-list may consist of a constant expression or an open brace, "C", followed by zero or more initializer-lists separated by commas, followed by a closing brace, ")". The constant expression is defined below in the paragraph on "Expressions"..

When the item to be initialized is a scalar, (char, int, long, float, pointer), the initializer may consist of only a single constant expression which may, optionally, be enclosed in braces, "(", ")".

For any item which is an aggregate, such as a structure or array, the initializer consists of an initializer-list enclosed in braces. The initial values are applied to each element of the structure or array in the order in which they appear. If fewer values appear than there are elements in an array or members in a structure, then the remaining elements or members are initialized to zero.

This definition may be applied recursively to aggregates of aggregates (sub-aggregates) so that the values of elements of sub-arrays and sub-structures may be explicitly defined. The symantics for subaggregate initialization are as follows:

If the initializer-list begins with a left brace, then the succeeding initializers, up to the next right brace, apply to the sub-aggregate. If a right brace is encountered before all the values of the sub-aggregate are initialized, the succeeding members of the sub-aggregate are initialized to zero. If the sub-aggregate initializer-list does not begin with a left brace, then as many elements from the initializer-list are used as is necessary to initialize all the members or elements of the sub-aggregate.

It is not permitted to initialize variables of type union.

In the case of an array in which the size is not specified, the Compiler will set the size of the array to the number of initialized values specified for it.

In the special case of a character array the initializer may take the form of a constant string. The array will be initialized such that each element of the array is set to the value of the corresponding character in the string constant. The terminating NULL is also considered part of the initializer and is encoded in the array. As above, if the size of the array is left unspecified the size will be the same as that of the NULL terminated string which initializes it.

The syntax for an initialized automatic variable is slightly different than for that of an external or static variable. It may consist of an equal sign, "=", followed by an expression which may, optionally, be enclosed in braces, "(", and ")". Notice that this definition allows an arbitrarily complex expression which may include constants, functions, and previously declared variables. The expression must evaluate to a scalar or float; it is not permitted to initialize aggregate (structure or array) automatic variables.

FUNCTION DEFINITIONS

A function definition is the mechanism by which a code segment is defined. Most programs include a function called "main" which is, by default, the function executed when the program starts. A function definition is indicated by an optional storage class specifier, followed by an optional type modifier, followed by an optional type specifier, followed by a declarator followed by a set of parentheses which enclose zero or more identifiers, followed by zero or more data declarations, followed by a compound statement. The storage class specifier may be any of the following.

```
extern
static
```

The type modifier may be any of the following.

```
long
short
```

```
          unsigned
```

The type may be any of the following.

```
          char
          int
          float
          <typename>
```

If the storage class is static, then the function will be known only in the program file in which it was defined; otherwise it will be known externally. If the storage class is omitted the function defaults to external. The type modifiers may be used only for functions whose type specifier is int or unspecified. The type specifiers in conjunction with the declarator form indicate the type of the function's return value. The type of the return value may only be char, int (long, short or unsigned), float, or pointer. If the type specifier is omitted it defaults to int.

ABSTRACT TYPE DECLARATIONS

There are two cases in which it may be necessary to refer to a data type without referring to any particular identifier. One of these cases involves the cast mechanism and the other involves the sizeof operator. In either case it may be necessary to specify an abstract type. An abstract type is indicated by an optional type modifier, followed by a type specifier, followed by an abstract declarator, where an abstract declarator is defined the same as a normal declarator above except that no identifier is permitted. That is, an abstract declarator may be a null sequence of characters, or an abstract declarator preceded by a star, or an abstract declarator followed by a set of brackets (which may contain a constant expression), or an an abstract declarator followed by an empty set of parentheses, or an abstract declarator enclosed in parentheses. In the last case the sequence of characters inside the parentheses may not be null. In the case of a cast, either the type modifier or the type specifier, but not both, may be omitted. If the type specifier is omitted int is assumed.

EXPRESSIONS

An expression is any construct which returns a value. The C language is very general about expressions. Expressions include constants, strings, identifiers which have been suitably declared, and expressions enclosed in parentheses. The result of any expression operation on an expression is also an expression. An expression may have side effects. This means, for example, that a variable may become changed in the process of evaluating an expression. This is typical of function calls but may also occur in some of the arithmetic expressions, as with the increment operator (x++) where the variable is incremented after its value is taken.

A string is in all cases treated like an array of characters. A string is the same syntactically as a character array identifier and thus is of type pointer to character when used in an expression.

Any expression may be enclosed in parentheses. The effect is to cause the enclosed expression to be completely evaluated before operators external to the parentheses are applied. The resultant type and value are that of the enclosed expression. The fact that an expression evaluates to an lvalue is not altered by enclosing such an expression in parentheses.

CONVERSIONS

The conversion of a value from one data type to another may be done explicitly, by using a cast for example, or may be implicitly carried out when some operation is performed, as when an integer is assigned to a float.

IMPLICIT CONVERSIONS

Many conversions are carried out automatically by the Compiler, particularly in the case of arithmetic expressions. The general pattern for deciding what will be converted to what in an arithmetic operation involving two operands is as follows:

> If either operand is of type float the other will be converted to float and that will be the resultant type;
> Otherwise if either operand is of type long int the other will be converted to long int and that will be the resultant type;
> Otherwise if either operand is of type unsigned int the other will be converted to unsigned int and that will be the resultant type;
> Otherwise if either operand is of type int the other operand will be converted to int and that will be the resultant type;
> Otherwise if either operand is of type short int the other operand will be converted to short int and that will be the resultant type;
> otherwise both operands must be of type char and that is the resultant type.

Notice that character expressions are not always automatically converted to integer and, in general, when used in arithmetic expressions, a character expression is converted to the type of the other operand. Thus, when two expressions of type character are added, the result will be of type character. If the result cannot fit in a character size space an overflow condition will occur. Character expressions are, however, always converted to integer when used as function parameters.

The following conventions apply to the results of various conversions. Note that Integral includes all types other than float.

Float to integral Type: The conversion from float to an integral type is as follows. The fractional part of the float is truncated to produce an integral value (truncation is always toward 0), and this is the resultant value if the truncated value is within the range which can be represented by the specified integral type. If the truncated value is larger than that which can be represented by the specified integral type, then the result is undefined.

Integral to Float Type: The conversion of an integral expression to type float results in the value of the integral expression as represented in floating point format. If the integral expression has more bits representing its value than the floating point allows in its mantissa, there will be some loss of precision when large numbers are converted. Presently this happens only when converting long integers to float.

Integral to Integral Type: if the bit length of the source expression type is longer than the bit length of the resultant type, then the only conversion done is to discard the excess high order bits. When the bit length of the destination type is longer than the bit length of the source expression type, excess high order bits will be filled with either the sign bit of the source expression or zeros. If the source expression is of unsigned type then high order bits are zero filled; otherwise they are sign filled. If both source expression type and destination type are the same length then no actual change in the bit pattern takes place.

EXPLICIT CONVERSIONS

Sometimes it is desired to force a conversion explicitly. This is called casting an expression from one type to another, and the mechanism by which this is done is called a cast. A cast is indicated by an expression preceded by a set of parentheses which enclose a type specifier followed by an abstract declarator (as described in the paragraph on abstract data declarations under DATA CONVENTIONS).

LVALUES

There is a distinction made between expressions which evaluate to constant values and those which evaluate to variable values. An expression which evaluates to a variable value is called an lvalue. Lvalues may be changed, whereas constant values may not. It makes no sense, for example, to place a constant value (a non-lvalue) to the left of an assignment operator because no new value may be assigned to it. Any attempt to do this will be flagged as an error by the Compiler. In fact, the "l" in the term "lvalue" is intended as a reminder that this value may be placed to the left of an assignment operator.

CONSTANT EXPRESSIONS

In certain cases Introl-C may require the use of a constant expression. The set of constant expressions is a subset of the set of regular expressions. Constant expressions are expressions which can be evaluated to a scalar at compile time and thus may contain no variables or floating point values. Likewise a constant expression may contain no operators which change the value of any of their operands or have variable results. The legal constant operators are the unary operators:

```
! ~ -  sizeof
```

the binary operators:

```
*  /  %  +  -   <<  >>   <  <=   >  >=   ==  !=  &  ^  |  &&  ||
```

and the trinary operator:
?:

In the case of a constant expression used as an initializer, the expression may alternatively consist of a floating point constant (possibly preceded by a negative sign), or an expression which evaluates to a constant pointer.

A constant pointer is one whose value is known at compile time. This includes function names, static and external array names, static and external variables which are preceded by the addressing operator, "&", or any of the above offset by a constant expression. The addresses of automatic variables are not permitted in such an expression because their location is dynamic (not known at compile time).

OPERATORS

The following is a list of operators in the order of their priority. Also listed is the order of evaluation of operators when two or more operators of the same priority appear in an expression.

OPERATOR	EVALUATED
() [] -> .	left to right
! ~ ++ -- - (<type>) * & sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= >>= <<= &= ^= |=	right to left
'	left to right

The operators are described below in the order of their priorities.

ADDRESSING OPERATORS

Addressing operators evaluate left to right.

Function Operator () The function operator is indicated by a pair of parentheses preceded by an expression which evaluates to type "function". There may optionally be a list of expressions separated by commas within the parentheses. The effect is to execute the function named. The result of the function operator is an expression which has a value of whatever type has been defined as the return type of the function. The expressions within the parentheses may be of any type and any number; no checking is done to verify that the types and number of the expressions within the parentheses in the function call agree with the types and number specified in the function declaration. Functions may be called recursively.

Array operator [] The array operator is indicated by an expression followed by a pair of brackets which contain an expression. One of the expressions must evaluate to type pointer while the other must evaluate to an integral type. It is usually considered a good programming practice to make the first expression (the one outside the brackets) the one which evaluates to type pointer. This is not of necessity, however, due to the fact that el[e2] is defined to be identical to *((el)+(e2)). Notice that addition is a commutative operator and, thus, so is the array operator. The result of an array operation is an expression which is of the type pointed to by the pointer expression. The array operator returns the value of the object that is pointed to when the integral value is multiplied by the size of the type pointed to and then added to value of the pointer. The effect is to return the value of the object which is displaced the integral number from the beginning of an array pointed to by the pointer.

Structure Member Operator. The structure member operator is indicated by an expression which evaluates to type structure, followed by a period, ".", followed by an identifier; as in "a.b". In Introl-C the expression must evaluate to a structure type which has the identifier as a legal member; otherwise, the Compiler will generate an error message. The result is an expression whose type and value is that of the indicated member in the structure.

Structure Member Pointer Operator -> The structure member pointer operator is indicated by an expression which evaluates to type pointer to structure followed by a dash-greater-than character combination, "->", followed by an identifier; as in "a->b" (there may be no white space between the dash and the greater than sign). In Introl-C the type of the structure pointed to by the expression must have the identifier as a legal member. The result is an expression whose type and value is that of the indicated member in the structure pointed to.

UNARY OPERATORS

Unary operators evaluate right to left.

Logical Not Operator ! The logical Not operator is indicated by an exclamation mark, "!", followed by an expression. The result is an expression whose type is character and whose value is 0 (zero) if the original expression was non-zero and 1 (one) otherwise.

Bitwise Not Operator ~ The bitwise not operator is indicated by a tilde, "~", followed by an expression. The result is an expression with a value equal to the one's complement of the original expression and with the same type as the original expression. The bitwise Not operator may not be applied to types pointer and float.

Increment Operator ++ The increment operator has two forms. It is indicated by a double plus (two successive plus signs with no intervening white space, "++") either immediately preceding or following an expression. The expression must evaluate to an lvalue (that is, a variable, something which can be written to). When the double plus precedes a variable, the variable is incremented by one and the resultant expression is the new value of the variable. When the double plus follows a variable, the variable is also incremented but the resultant expression is the value the variable had before it was incremented. When the increment operator is applied to a pointer, the pointer is incremented by the length of the object to which it points; thus it will point to the next object in sequence.

Decrement Operator -- The decrement operator (like the increment operator) has two forms. It is indicated by a double minus (two successive minus signs with no intervening white space, "--") either immediately preceding or following an expression. The expression must evaluate to an lvalue (that is, a variable, something which can be written to). When the double minus precedes the variable the variable is decremented by one and the resultant expression is the new value of the variable. When the double minus follows the variable, the variable is also decremented but the resultant expression is the value the variable had before it was decremented. When the decrement operator is applied to a pointer the pointer is decremented by the length of the object to which it points; thus it will point to the previous object in sequence.

Unary Minus Operator - The unary minus operator is indicated by a minus sign, "-", followed by an expression. The resultant expression is the algebraic negation of the original expression. The action of the unary minus is undefined when used on types unsigned integer and character (which is also unsigned).

Cast Operator (type) The cast operator is indicated by a data type name in parentheses, followed by an expression. A data type name is like a data type declaration but without the object to which it would normally refer. For example, to cast some expression to type "function returning pointer to character", one would type "(char *())El" (where El is an expression). The expression may be of any type. The resultant expression has the type specified by the cast.

Indirection Operator * The indirection operator is indicated by a star, "*", followed by an expression which must be of type pointer. The resultant expression has the type and value of the object to which the pointer points.

Address Operator & The address operator is indicated by an ampersand, "&", followed by an lvalue. The resultant expression is a pointer to the object indicated by the lvalue.

Size of Operator sizeof The size of operator is indicated by the keyword, "sizeof", followed by either a type name enclosed in parentheses, or an expression. The result is an expression of type integer whose value is the size, in bytes, of an object of the type indicated.

MULTIPLICATIVE OPERATORS

Multiplicative operators evaluate left to right.

Multiplication Operator * The multiplication operator is indicated by an expression, followed by a star, "*", followed by an expression. The result is an expression whose value is that of the algebraic multiplication of the two expressions.

Division operator / The division operator is indicated by an expression, followed by a slash, "/", followed by an expression. The result is an expression whose value is that of the algebraic division of the first expression by the second. If both of the expressions are of integral type then the result will also be of integral type and any fractional result will be discarded.

Modulo Operator % The modulo operator is indicated by an expression, followed by a percent symbol, "%", followed by an expression. The result is an expression whose value is the first expression modulo the second expression. That is, the first expression is integer divided by the second expression with the result equal to the remainder. Both expressions must be of integral type.

ADDITIVE OPERATORS

Additive operators evaluate left to right.

Addition Operator + The addition operator is indicated by an expression, followed by a plus symbol, "+", followed by an expression. The result is an expression whose value is the algebraic sum of the expressions.

Subtraction Operator - The subtraction operator is indicated by an expression, followed by a minus sign, "-", followed by an expression. The result is an expression whose value is the algebraic result of the second expression subtracted from the first expression.

SHIFT OPERATORS

Shift operators evaluate left to right.

Left Shift Operator << The left shift operator is indicated by an expression, followed by a double less-than symbol, "<<", followed by an expression. The result is an expression whose value is that of the first expression after having been bitwise left shifted by the number of bits indicated by the second expression. Zeros are shifted into the low order bit positions. Both expressions must be of integral type.

Right Shift Operator >> The right shift operator is indicated by an expression, followed by a double greater-than symbol, ">>", followed by an expression. The result is an expression whose value is that of the first expression after having been bitwise right shifted by the number of bits indicated by the second expression. If the first expression is of signed type, its sign bit will be shifted into the high order bit positions; otherwise zeros will be shifted into the high order bit positions. Both expressions must be of integral type.

RELATIONAL OPERATORS

Relational operators evaluate left to right.

Less-Than Operator < The less-than operator is indicated by an expression, followed by a less-than symbol, "<", followed by an expression. The result is an expression of type character which has a non-zero (true) value if the first expression is algebraically less than the second expression, and a zero (false) value otherwise.

Less-Than Equal Operator <= The less-than equal operator is indicated by an expression, followed by a less-than equal character combination, "<=", followed by an expression. There may be no white space between the less-than symbol and the equal symbol. The result is an expression of type character which has a non-zero (true) value if the first expression is algebraically less than or equal to the second expression, and a zero (false) value otherwise.

Greater-Than Operator > The greater-than operator is indicated by an expression, followed by a greater than symbol, ">", followed by an expression. The result is an expression of type character which has a non-zero (true) value if the first expression is algebraically greater than the second expression, and a zero (false) value otherwise.

Greater-Than Equal operator >= The greater-than equal operator is indicated by an expression, followed by a greater-than equal character combination, ">=", followed by an expression. There may be no white space between the greater-than symbol and the equal symbol. The result is an expression of type character which has a non-zero (true) value if the first expression is algebraically greater than or equal to the second expression, and a zero (false) value otherwise.

EQUALITY OPERATORS

Equality operators evaluate left to right.

Equal To Operator == The equal-to operator is indicated by an expression, followed by a double equal sign, "==", followed by an expression. There may be no white space between the two equal signs. The result is an expression of type character which has a non-zero (true) value if the first expression is algebraically equal to the second expression, and a zero (false) value otherwise.

Not Equal Operator != The not-equal operator is indicated by an expression, followed by an exclamation mark equal character combination, "!=", followed by an expression. There may be no white space between the exclamation mark and the equal sign. The result is an expression of type character which has a non-zero (true) value if the first expression is algebraically unequal to the second expression and a zero (false) value otherwise.

BITWISE AND
The bitwise And operator evaluates left to right.

Bitwise And Operator & The bitwise And operator is indicated by an expression, followed by an ampersand, "&", followed by an expression. The result is an expression whose value is the bitwise And of the two expressions. Both expressions must be of integral type.

BITWISE EXCLUSIVE OR
The bitwise exclusive Or operator evaluates left to right.

Bitwise Exclusive Or operator - The bitwise exclusive or operator is indicated by an expression, followed by a caret, "-", followed by an expression. The result is an expression whose value is the bitwise exclusive Or of the two expressions. Both expressions must be of integral type.

BITWISE OR
The bitwise Or operator evaluates left to right.

Bitwise Or Operator | The bitwise Or operator is indicated by an expression, followed by a vertical bar, "|", followed by an expression. The result is an expression whose value is the bitwise Or of the two expressions. Both expressions must be of integral type.

LOGICAL AND
The logical And operator evaluates left to right.

Logical And operator && The logical And operator is indicated by an expression, followed by a double ampersand, "&&", followed by an expression. The result is an expression of type character which has a non-zero (true) value if both expressions had non-zero values, and a zero (false) value otherwise. All Logical-And expressions are evaluated in short circuit mode. That is, the expression is evaluated left to right and, if the first expression has a zero value, then the second expression is not evaluated.

LOGICAL OR
The logical Or operator evaluates left to right.

Logical Or Operator || The logical or operator is indicated by an expression, followed by double vertical bars, "||", followed by an expression. The result is an expression of type character which has a non-zero (true) value if either of the expressions has a non-zero value, and a zero (false) value otherwise. All Logical-Or expressions are evaluated in short circuit mode. That is, the expression is evaluated left to right and, if the first expression has a non-zero value, then the second expression is not evaluated.

CONDITIONAL EXPRESSION

The conditional expression evaluates right to left.

Conditional operator ?: The conditional expression operator, a trinary operator, is indicated by an expression, followed by a question mark, "?", followed by an expression, followed by a colon, ":", followed by an expression. If the first expression evaluates to a non-zero value, the second expression is evaluated; otherwise the third expression is evaluated. If the second and third expressions are of different type, the usual arithmetic conversion conventions are applied to make the types identical. The resultant expression has the same type and value as the evaluated expression.

ASSIGNMENT OPERATORS

Assignment operators evaluate right to left.

Assignment Operator = The assignment operator is indicated by an lvalue, followed by an equal sign, "=", followed by an expression. The lvalue's old value will be replaced by the value of the expression. The result is an expression with a type and value the same as that of the lvalue.

Update Assignment 0perator <binary operator >= The update assignment operator is indicated by an lvalue, followed by a binary operator-equal sign character combination (for example +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, or |=), followed by an expression. There may be no white space between the binary operator and the equal sign. The effect of

```
    <lvalue> op= <expression>
```

is identical to

```
    <lvalue> = <lvalue> op <expression>
```

except that the lvalue is evaluated only once. The result is an expression with the same value and type as that of the lvalue.

COMMA

The comma operator evaluates left to right.

Comma Operator , The comma operator is indicated by an expression, followed by a comma, ",", followed by an expression. Each expression is evaluated from left to right. The resultant expression has the type and value of the second expression.

STATEMENTS

Statements include the set of all expressions along with various constructs which control program flow. Statements are executed sequentially unless the program flow has been altered by one of the program flow control statements.

EXPRESSION STATEMENT

Any expression may be used as a statement if it is terminated by a semicolon. The resultant value of the expression has no effect. Presumably the expression will have some side effect, such as altering a memory location as is done in an assignment expression.

An expression statement which has no side effects is flagged as an error by the Compiler.

COMPOUND STATEMENT

A compound statement, also called a block, consists of a left brace, "(", followed by zero or more data declarations, followed by zero or more statements, followed by a right brace, ")". A block has the effect of "bracketing" a group of statements so that they become, for syntactical purposes, a single statement. Thus the compound statement may be used anywhere any other statement may be used. All data declared inside the block is local to the block unless specified as being external.

CONDITIONAL STATEMENT

The conditional statement has two forms. One form is the following: the keyword "if", followed by a set of parentheses containing an expression, followed by a statement. The expression is evaluated and, if its resultant value is non-zero, then the statement will be executed; otherwise it will not be executed. The other form of the conditional statement consists of the keyword "if", followed by a set of parentheses containing an expression, followed by a statement, followed by the keyword "else", followed by a statement. The expression is evaluated and, if its resultant value is non-zero, then the first statement is executed; otherwise the second statement is executed.

WHILE STATEMENT

The while statement is indicated by the keyword "while", followed by a set of parentheses containing an expression, followed by a statement. The expression will be evaluated repeatedly until it evaluates to a zero value with the statement being executed after each non-zero evaluation of the expression. If the expression evaluates to zero initially, then the statement will not be executed at all.

DO STATEMENT

The do statement is indicated by the keyword "do", followed by a statement, followed by the keyword "while", followed by a set of parentheses containing an expression. The statement is executed repeatedly, with the expression being evaluated after each execution of the statement, until the expression evaluates to zero. The statement is always executed at least once.

FOR STATEMENT

The for statement is indicated by the keyword "for", followed by an open paren, "(", followed by an optional expression, followed by a semicolon, ";", followed by an optional expression, followed by a semicolon, ";", followed by an optional expression, followed by a close paren, ")", followed by a statement. The first expression will be evaluated exactly once. The second expression will be evaluated repeatedly until it evaluates to a zero value, with the statement being executed and the third expression being evaluated after each non-zero evaluation of the second expression. Notice that all three of the expressions are optional. If the second expression is omitted it will be assumed to be an expression which always evaluates to a 1, thus making the for loop execute forever. The effect of omitting the first or the third expression is simply that there will be nothing to evaluate in their respective positions.

SWITCH STATEMENT

The switch statement is indicated by the keyword "switch", followed by an expression enclosed in parentheses, followed by a statement. The expression is evaluated and cast to type integer. The resultant value is then matched against any case labels in the statement portion of the switch. If a match is found, execution will be resumed at the location where the case label was defined. If no match is found, but there is a default prefix in the statement portion of the switch statement, then execution will continue at the location following the default prefix; otherwise no part of the statement portion of the switch will be executed.

CASE LABEL STATEMENT

The case label may only appear in the statement portion of a switch statement. It is indicated by the keyword "case", followed by a constant expression, followed by a colon ":", followed by a statement. Its effect is to mark the statement as a possible entry point in a switch statement.

DEFAULT STATEMENT

The default statement may only appear in the statement portion of a switch statement. It is indicated by the keyword "default", followed by a colon, ":", followed by a statement. Its effect is to mark the statement as the default entry point in a switch statement. This entry is taken when none of the case labels matches the expression in the switch statement. The default statement may appear no more than once in any given switch statement.

BREAK STATEMENT

The break statement is indicated by the keyword "break", followed by a semicolon, ";". The break statement causes termination of the smallest enclosing while, do, for, or switch statement. Control passes to the statement following the terminated statement.

CONTINUE STATEMENT

The continue statement is indicated by the keyword "continue", followed by a semicolon, ";". The continue statement is permitted only in while, do, and for statements. In each of these statements the continue statement causes immediate completion of the statement portion of the above mentioned looping statements. The effect is that the current iteration of the looping statement terminates and execution continues at the point in the looping statement which is normally executed when the loop completes an iteration.

RETURN STATEMENT

The return statement is indicated by the keyword "return", optionally followed by an expression, followed by a semicolon, ";". The return statement causes a function to return control to its caller. If the optional expression is included, it will be evaluated and its value will be the return value of the function; otherwise the function's return value is undefined. The return statement is optional; there is an implicit "return" statement at the end of every function body.

GOTO STATEMENT

The goto statement is indicated by the keyword "goto", followed by an identifier followed by a semicolon, ";", where the identifier is a label appearing on a label statement which exists in the same function as the goto statement. The goto statement causes control to be transferred to the statement marked by the label identifier. The target label must appear in the same function as the goto.

LABEL STATEMENT

The label statement is indicated by an identifier, followed by a colon, ":", followed by a statement. Its effect is to mark a statement as a possible destination for a goto statement.

NULL STATEMENT

The null statement is indicated by a lone semicolon, ";". It has no effect except to take up the place of a statement. It may be placed anywhere a statement is permitted.

APPENDICES

This section contains miscellaneous reference information which may be useful to the programmer.

APPENDIX A

INTROL-C / STANDARD C

The following differences exist between Introl-C and "standard C" as it is defined in the Kernighan and Ritchie book, "The C Programming Language".

OMMISSIONS

1) The current release of Introl-C does not support fields.

2) The current release of introl-C does not support the double data type.

3) The current release of Introl-C does not support the #line and #if preprocessor directives (all other directives, including #ifdef and #ifndef, are supported, however).

EXTENSIONS

4) Nesting of comments is permitted in Introl-C. Thus large sections of code may be "commented out" by simply bracketing the code segment with /* and */.

5) Introl-C provides separate name spaces for all structure and union member names, allowing the use of identical names in different struct and union declarations.

6) Introl-C does not permit the use of the obsolete assignment-update operator in which the operator follows the equal sign. Thus x=-l is not identical to x-=l in Introl-C as it may be in some other implementations of C.

7) Introl-C permits symbols to up to 90 characters in length.

APPENDIX B

DATA TYPE CONVERSIONS

The following describes the result of all conversions, implicit or otherwise.

<u>char to float:</u> The conversion of a character to type float results in the value of the character being represented in floating point format. Characters are unsigned quantities.

<u>char to int:</u> Characters are converted to integers by padding zeros on the left. In present versions of introl-C characters are unsigned.

<u>char to long int:</u> Characters are converted to long integers by padding zeros on the left.

<u>char to short int:</u> Characters are converted to short by padding zeros on the left.

<u>char to unsigned int:</u> Characters are converted to unsigned by padding zeros on the left.

<u>char to pointer:</u> Characters are converted to pointer by padding zeros on the left.

<u>float to char:</u> The fractional part of the float is truncated to produce an integral value (truncation is always toward 0). This is the resultant value if the value is within the range which can be represented by a character. If the value is larger than that which can be represented by a character, then the result is the maximum value possible for a character. If the value is smaller than that which can be represented by a character, the result is set to the minimum value possible for a character.

<u>float to int:</u> The fractional part of the float is truncated to produce an integral value (truncation is always toward 0). This is the resultant value if the value is within the range which can be represented by a signed integer. If the value is larger than that which can be represented by an integer, then the result is the maximum value possible for an integer. If the value is smaller than that which can be represented by an integer, the result is set to the minimum value possible for an integer.

<u>float to long int:</u> The fractional part of the float is truncated to produce an integral value (truncation is always toward 0). This is the resultant value if the value is within the range which can be represented by a long integer. If the value is larger than that which can be represented by a long integer, then the result is the maximum value possible for a long integer. If the value is smaller than that which can be represented by a long integer, the result is set to the minimum value possible for a long integer.

float to short int: The fractional part of the float is truncated to produce an integral value (truncation is always toward 0). This is the resultant value if the value is within the range which can be represented by a short integer. If the value is larger than that which can be represented by a short integer, then the result is the maximum value possible for a short integer. If the value is smaller than that which can be represented by a short integer, the result is set to the minimum value possible for a short integer.

float to unsigned int: The fractional part of the float is truncated to produce an integral value (truncation is always toward 0). This is the resultant value if the value is within the range which can be represented by an unsigned integer. If the value is larger than that which can be represented by an unsigned integer, then the result is the maximum value possible for an unsigned integer. If the value is smaller than that which can be represented by an unsigned integer the result is set to the minimum value possible for an unsigned integer.

float to pointer: The fractional part of the float is truncated to produce an integral value (truncation is always toward 0). This is the resultant value if the value is within the range which can be represented by a pointer. If the value is larger than that which can be represented by a pointer, then the result is the maximum value possible for a pointer. If the value is smaller than that which can be represented by a pointer, the result is set to the minimum value possible for a pointer.

int to char: Integers are converted to characters by truncating the excess high order bits.

int to float: The conversion of an integer to type float results in the value of the integer represented in a floating point format.

int to long int: Integers are converted to long integers by sign extension.

int to short int: Integers are converted to short integers by truncating any excess high order bits.

int to unsigned int: The conversion from integer to unsigned integer is conceptual and no actual change in the bit pattern takes place. Thus the value of a positive integer converted to unsigned integer does not change while the value of a negative integer appears as a large unsigned integer.

int to pointer: The conversion from integer to pointer is conceptual and no actual change in the bit pattern takes place.

long int to char: Long integers are converted to type character by truncating the excess high order bits.

long int to float: The conversion of a long integer to type float results in the value of the long integer represented in floating point format. There may be some loss of precision for large values because the number of bits used to represent the long (31 not including sign) is larger than the number of bits used to represent the mantissa of the float (24).

long int to int: Long integers are converted to type integer by truncating any excess high order bits.

long int to short int: Long integers are converted to short integers by truncating the excess high order bits.

long int to unsigned int: Long integers are converted to unsigned integers by truncating the excess high order bits.

long int to pointer: Long integers are converted to pointer by truncating the excess high order bits.

short int to char: Short integers are converted to character by truncating any excess high order bits.

short int to float: The conversion of a short integer to type float results in the value of the short represented in floating point format.

short int to int: When short integers are converted to type integer, any excess high order bit positions in the result are filled by sign extending the short integer.

short int to long int: When short integers are converted to type long integer, any excess high order bit positions in the result are filled by sign extending the short integer.

short int to unsigned int: When short integers are converted to type unsigned integer, any excess high order bit positions in the result are filled by sign extending the short integer.

short int to pointer: When short integers are converted to pointer, any excess high order bit positions in the result are filled by sign extending the short integer.

unsigned int to char: Unsigned integers are converted to type character by truncating the excess high order bits.

unsigned int to float: The conversion of an unsigned integer to type float results in the value of the unsigned integer represented in floating point format.

unsigned int to int: The conversion from unsigned integer to integer is conceptual and no actual change in the bit pattern takes place. Thus, when an unsigned integer with a value greater than the maximum integer value is converted to an integer, the result appears as a negative number.

unsigned int to long int: Unsigned integers are converted to long by padding zeros on the left.

unsigned int to short int: Unsigned integers are converted to type short integers by truncating any excess high order bits.

unsigned int to pointer: The conversion from unsigned to pointer is conceptual and no actual change in the bit pattern takes place.

pointer to char: Pointers are converted to type character by truncating the high order bits.

pointer to float: The conversion of a pointer to type float results in the value of the pointer as represented in floating point format. The value of a pointer is interpreted as an unsigned quantity.

pointer to int: The conversion from pointer to integer is conceptual and no actual change in the bit pattern takes place.

pointer to long int: Pointers are converted to type long integer by padding the high order bits with zeros.

pointer to short int: Pointers are converted to short integer by truncating any excess high order bits.

pointer to unsigned int: The conversion from pointer to unsigned integer is conceptual and no actual change in the bit pattern takes place.

APPENDIX C

INTROL-C/6809 COMPILER DATA, REGISTER USAGE. AND PARAMETER PASSING CONVENTIONS

DATA

The value of char data is represented in an eight bit (one byte) memory location. A char is an unsigned small integer that can contain a value from zero to 255.

Int variables are contained in two bytes (16 bits) and represent a two's complement value that may be in the range -32768 to +32767.

All signed integers are represented in two's complement form.

Short is a synonym for int in this implementation.

Unsigned (or unsigned int) variables are contained in two bytes (16 bits) and may contain values in the range 0 to 65535.

Long (or long int) variables are contained in four bytes (32 bits) and contain values in the range -2147483648 to 2147483647.

Floats are contained in four bytes (32 bits) and contain values as defined by the IEEE standard for 32 bit floating point numbers. (See also the discussion on floats in the "Definition of Introl-C" section of this manual.)

A structure has a size exactly equal to the sum of the sizes of its parts. There are no unused spaces in structures. For example the structure declaration:

```
struct
      {
      int a;
      char b;
      unsigned d;
      char e[2];
      long f;
      float g;
      } f;
```

will create the following memory allocation (assume the byte numbers represent offsets from the beginning of structure f)

Byte	Contents
0,1	int value of member a. (Byte 0 is the high byte.)
2	Char value of member b.
3,4	Unsigned value of member d.
5	e[0]
6	e[1]

7,8, 9, 10	Long int value of member f. (Byte 7 is the high byte.)
11,12,13,14	The first, most significant bit of the first byte is the sign of the float. The next seven bits of the first byte and the first bit of the next byte comprise the biased exponent. The remaining 23 bits comprise the mantissa and make up the remainder of the second byte as well as the next two bytes.

A union is the size of its largest member. All unions pack towards the left. This means that a char variable coexisting with an int in a union will actually be allocated the byte representing the high byte of the integer's value.

An array has the size of one of its elements multiplied by the given dimension of the array. An array declaration such as:

```
          char a[10];
```

defines "a" to be a character array with ten elements and therefore ten bytes long.

REGISTER USAGE

The 6809 has two eight bit accumulators (usable as a single sixteen bit register), three general purpose index registers, a hardware stack pointer and a program instruction counter. These registers are allocated by the Compiler as follows.

The B accumulator is used as the char accumulator for arithmetic expressions that involve char values. The D register (A:B) is used as the int and unsigned accumulator. A programmer is free to destroy these registers in a user written assembly language function. The B register is used to return character data from a function; the D register is used to return int, or unsigned values; and both the U and D registers are used to return long int. or float, with U containing the most significant half of the number.

The X, Y, and U registers are used in addressing operands. The contents of the X and U register may be destroyed by an assembly language routine without adverse effect. The Y register may also be modified, but only if the user is not generating position independent code. When generating position independent code, the Compiler assumes the Y register will in all cases contain the address of the beginning of its external and static data area. In such case, a program initialization routine must initialize the Y register before the first call to "main()".

The hardware stack pointer (SP) should be preserved through a function. The SP points to an area of read/write memory that has several uses: (1) The stack area is used to preserve a record of the execution history of the program, so that a function always "knows" who called and can return to the same place; (2) the stack is used to save the state of the processor in the event of an interrupt; (3) the stack is used to pass parameters to a function: and (4) the stack is used to allocate local variable space for a function. These first two functions of the stack are determined by the 6809 hardware and can be pursued further, if desired, by obtaining a reference book on the microprocessor. The third and fourth functions of the stack (parameter passing and local variable allocation) are described in the following paragraphs.

PARAMETER PASSING CONVENTIONS

When a function is called in this implementation the second through the last parameters are pushed on the stack in reverse order (last parameter first). The first parameter is loaded into the D accumulator. If the first parameter is a long or float, the high order word is loaded into the U register. Char values are converted to int when passed as a parameter. Either the jump to subroutine (JSR) or the long branch to subroutine (LBSR) instruction is then used to call the desired function. After the function returns, the area in the stack used for parameters is freed. The return value of the function is assumed to be in the U and D registers, where U is assumed to hold the most significant 16 bits of a returned long or float value while the D register holds the least significant 16 bits. Integer-sized data is returned in the D register. Character data is returned in the low order 8 bits of the D register (the B register). When returning character type data, it is a good idea to clear the upper 8 bits of the D register (the A register).

A function call such as:

```
            f(a,b,1+2)
```

would generate the 6809 code with the following meaning:

```
            push   (the value of 1+2)
            push   (the value of variable b)
            load   (the value of  variable a)
            LBSR    f
            deallocate 4 bytes from the SP (total pushed
                                            parameter size)
```

When the function is entered, the stack frame looks like this:

```
                          Stack Contents                  Offset
                    other data on the stack                 SP+6
                    the value of 1+2                        SP+4
                    the value of variable b                 SP+2
            SP ->   return address                          SP+0

            D =  value of variable a
```

LOCAL DATA

If a function needs auto storage locations it allocates them below the return address of the stack frame described above. Suppose the function f() has the following declaration:

```
    f(x,y,z)
          int x,y,z;
          {
          char a;
          int  b;
          .
          .
          .
```

The function would expect its parameters to be in the stack frame as described above. The function will often save parameter 1 (passed in the D register) in the stack just under the return address. After entering the function, the stack pointer would be modified to allow the storage of a and b below the return address of the stack frame. The new stack frame would look like this:

```
                         Stack Contents             offset
                    other data on the stack          SP+11 ...
                    the value of parameter z         SP+9
                    the value of parameter y         SP+7
                    return address                   SP+5
                    the value of parameter x         SP+3
                    variable b                       SP+l
          SP ->     variable a                       SP+0
```

Note that char variables use only one byte as auto variables. The only time they are automatically given two bytes is when passed as parameters. The function has the responsibility of "cleaning up" after itself by removing the allocation of variables a and b from the stack. Allocating memory from the stack is accomplished by subtracting the desired number of bytes from the SP and using the area between the new SP and the old SP. Deallocating memory from the stack is the opposite: add the number of bytes to deallocate to the SP.

There are two important things to remember about the stack pointer. The first is that it must always point to the return address of the caller when the function is complete. The second is that the stack pointer must always point to an area of memory large enough to hold all the auto variables of a series of functions at their deepest nesting level, allow room for the parameters and return addresses, leave space for any temporary variables that might be used on the stack, and allow room for saving the system state if the programs are to be run in an interrupt environment. In other words, the stack is very busy so make the stack area big enough!

INDEX

FC6809 INTROL-C

STANDARD LIBRARY
REFERENCE MANUAL

(FLEX)

Introl Corp.
647 W. Virginia St.
Milwaukee, WI 53204 USA

tel. (414) 276-2937

FC6809 STANDARD LIBRARY

This manual describes each of the standard library routines supported by the FC6809 Introl-C Standard Library. The FC6809 Standard Library is usable with the Introl "fld' Loader for producing programs that are compatible with, and executable under, the Flex operating system. Note that Introl-C uses system call names which may differ from those used by your operating system. Those system calls which perform a function which is analogous to a recognized UNIX system call have been given the corresponding UNIX name rather than the name used by the particular operating system. The library functions appear in alphabetical order in this-manual.

> IMPORTANT NOTE: The majority of functions contained in the Standard Library have been pre-assigned a module "class number" of zero (0). Several "non-zero" class Standard Library modules are also included for user convenience, however, and are identified in the Appendix at the end of this Standard Library Manual. In general, these non-zero class modules are alternate forms of identically named class zero modules that exist in the library, modified to fit specific programming applications.

The following is a list of the functions included in this manual.

NAME
     abs - integer absolute value

SYNOPSIS
```
     int     abs(i)
     int     i;
```

DESCRIPTION
     abs returns the absolute value of its integer operand.

DIAGNOSTICS

SEE ALSO

NOTES

NAME
    alloc - allocate memory

SYNOPSIS

```
     char    *alloc(size)
     int     size;
```

DESCRIPTION
    alloc will attempt to allocate a block of memory whose size is given by the argument. If it is successful it returns a pointer to that memory otherwise it returns NULL.

DIAGNOSTICS
    Returns NULL if the memory could not be allocated.

SEE ALSO
    free(), sbrk()

NOTES
    Alloc is an obsolete name for malloc(). It simply calls malloc() and returns.

NAME

atof - convert string to float

SYNOPSIS

```
float   atof(cptr)
char    *cptr;
```

DESCRIPTION

The atof function converts a string into a float which is then used as the return value of the function. The string should be null terminated although atof will stop reading the string as soon as an illegal character is reached. After ignoring preceding blanks the atof routine will convert as much of the string as conforms to normal floating point constant format to a floating point number. It will stop at the first character which is inconsistent with that format. If no floating point constant is found a 0 is returned.

A floating point constant consists of an integer part, a decimal point, a fractional part, and an exponential part. The integer and fractional parts may each consist of a string of one or more digits. The exponential part consists of an 'e' or 'E', followed by an optionally signed integer exponent. Either the integer or the fractional part (but not both) may be missing; either the decimal point or the exponential part (but not both) may be missing.

DIAGNOSTICS

SEE ALSO

atoi(), atol()

NOTES

Presently it is permitted to have spaces between the 'e' or 'E' and the first character of the integer representing the exponent.

NAME

atoi - convert string to integer

SYNOPSIS

```
int     atoi(ptr)
char    *ptr;
```

DESCRIPTION

Atoi's argument is a pointer to char which is assumed to point to a null terminated string which contains the ASCII representation of some integer number. The atoi function converts a string into an int which is the return value. The string should be null terminated although atoi will stop reading the string as soon as an illegal character is reached. After ignoring preceding blanks the atoi routine will convert as much of the string as conforms to normal integer constant format to an integer number. It will stop at the first character which is inconsistent with that format. If no integer constant is found a 0 is returned.

The integer constant format consists of an optional sign, followed by one or more digits. There should be no spaces interspersed within the number.

DIAGNOSTICS

SEE ALSO

atof(), atol()

NOTES

NAME

atol - convert string to long,

SYNOPSIS

```
long    atol(cptr)
char    *cptr;
```

DESCRIPTION

The atol function converts a string into a long which is the return value. The string should be null terminated although atol will stop reading the string as soon as an illegal character is reached. After ignoring preceding blanks the atol routine will convert as much of the string as conforms to normal long integer constant format to a long integer. It will stop at the first character which is inconsistent with that format. If no long integer constant is found a 0 is returned.

The long integer constant format consists of an optional sign, followed by one or more digits. There should be no spaces interspersed within the number.

DIAGNOSTICS

SEE ALSO

atof(), atoi()

NOTES

NAME

cprep - prepare environment for C program

SYNOPSIS

```
     int     cprep(argc,argv,eext)
     int     argc;
     char    **argv;
     char     *eext;
```

DESCRIPTION

Cprep first prepares the environment for the user C program and then call s "main", the usual entry-point to a user program. Cprep is usually referenced only from "cstart". The user program is not expected to make any explicit reference to this routine.

DIAGNOSTICS

SEE ALSO

cstart

NOTES

The result of an explicit reference to cprep is undefined.

NAME

cstart - runtime preparation routine

SYNOPSIS

DESCRIPTION

Cstart is a runtime preparation routine which is normally the first routine executed by an Introl-C program. Its only function is to set up the environment enough to allow the function "cprep" to be called. Cprep is a function which produces the runtime environment which is-expected by the user program. Cstart is included automatically by the linker. It is NOT expected that a user program will reference cstart explicitly via a function call.

DIAGNOSTICS

SEE ALSO

cprep()

NOTES

The result of an explicit reference to cstart is undefined.

NAME

ecvt - float to string conversion

SYNOPSIS

```
char    *ecvt(arg,ndigits,decpt,sign)
float   arg;
int     ndigits;
int     *decpt,*sign;
```

DESCRIPTION

This is a formatting routine used by printf for formatting floating point numbers in the e format.

Ecvt returns a pointer to a string which contains ascii characters representing a floating point number. The first argument is converted to a string whose length is indicated by the second argument. The third argument points to a variable in which the routine will write the location of the decimal point relative to the start of the string (negative numbers indicate that the decimal point is to the left of the first character of the string). The variable pointed to by the fourth argument is set nonzero if the float is negative otherwise it is set to zero.

The string is written in a static data area local to ecvt and is overwritten with the next call.

If the argument passed to ecvt is a legal floating point number the string will consist of a series of ascii digits terminated by a null. If the argument is out of the legal range for floats (as per the IEEE standard) the string will contain "NaN" (Not a Number). If the argument is either greater than the maximum or less than the minimum allowed for a float the characters "inf" (infinity) will be placed in the string (the fourth argument is set to indicate positive or negative infinity). The string itself contains neither a minus sign nor a decimal point nor a base ten exponent.

DIAGNOSTICS

SEE ALSO

fcvt(), itoa()

NOTES

NAME
     execl - execute a program

SYNOPSIS
```
     int     execl(cmd,arg0,arg1,...,0)
     char    cmd,*arg0,*arg1,.....;
```

DESCRIPTION
     Execl causes the present program to cease execution and a new program to execute. The name of the file to be executed must be contained in a string pointed to by the first argument. The additional arguments are assumed to be pointers to null terminated strings. These pointers will be passed to the program to be executed if they appeared as parameters on a command call line. The last argument MUST be a zero. The new process is given the arguments which follow the first argument in the execl call. The second argument of the execl call is the FIRST argument passed to the program to be executed (by convention referred to as argv(0). The last argument in the execl call <u>must</u> <u>always</u> <u>be</u> <u>a</u> <u>zero</u>.

DIAGNOSTICS
     This function NEVER returns.

SEE ALSO

NOTES
     The sum total of lengths of the argument strings (including a space to be placed between each argument) must not exceed the length of a FLEX line buffer, which is 128 bytes long.

NAME

exit - exit a program with file cleanup

SYNOPSIS

```
int     exit(stat)
int     stat;
```

DESCRIPTION

Exit aborts a C program and returns to the operating system. The status value is returned to the operating system. Exit also flushes any open file buffers and closes all open files before exiting.

DIAGNOSTICS

SEE ALSO

_exit()

NOTES

```
NAME
     _exit - exit a program without file cleanup

SYNOPSIS
     int     _exit(stat)
     int     stat;

DESCRIPTION
     _exit aborts a C program and returns to the  operating  system.
     The status value is returned to the operating system. The _exit
     routine does not explicitly flush the file buffers.

DIAGNOSTICS

SEE ALSO
     exit()

NOTES
```

NAME
     _extend - extend float

SYNOPSIS
```
     int     _extend(f,ef)
     float   f;
     struct  extflt
             {
             char    sign;
             int     exp;
             long    mantissa
             }  *ef;
```

DESCRIPTION
     _extend extends a floating point number (its first argument)
     and stores the result in the structure pointed to by the second
argument. The first element of the structure contains the sign
     bit of the number, the second element contains the unbiased
     exponent, and the thirs element contains the mantissa.

DIAGNOSTICS

SEE ALSO
     _unext()

NOTES

NAME

fclose - close file

SYNOPSIS

```
#include  "stdio.h"
int    fclose(fp)
FILE   *fp;
```

DESCRIPTION

Fclose will close the file indicated by its argument. The argument must be a file pointer which was previously returned from an fopen unless it is STDIN, STDOUT, or STDERR. If the file has been opened for writing, fclose will automatically flush the remaining contents of the buffer.

DIAGNOSTICS

fclose will return ERROR if the file could not be closed. The external variable "errno" will contain the error code which was returned by the operating system..

SEE ALSO

fgets(), fopen(), fprintf(), fputs(), fscanf(), getc()

NOTES

NAME

fcvt - float to string conversion

SYNOPSIS

```
char   *fcvt(arg,ndigits,decpt,sign)
float  arg;
int    ndigits;
int    *decpt,*sign;
```

DESCRIPTION

This is a formatting routine used by printf for formatting floating point numbers in the f format. It is similar to the "ecvt" routine except that the correct digit will be rounded as demanded by Fortran F-format for the number of digits indicated by the second argument

Fcvt returns a pointer to a string which contains ascii characters representing a floating point number. The first argument is converted to a string whose length is indicated by the second argument. The third argument points to a variable in which the routine will write the location of the decimal point relative to the start of the string (negative numbers indicate that the decimal point is to the left of the first character of the string). The variable pointed to by the fourth argument is set nonzero if the float is negative; otherwise it is set to zero.

The string is written in a static data area local to fcvt and is overwritten with the next call.

If the argument passed to fcvt is a legal floating point number the string will consist of a series of ascii digits terminated by a null. If the argument is out of the legal range for floats (as per the IEEE standard) the string will contain "NaN' (Not a Number). If the argument is either greater than the maximum or less than the minimum allowed for a float the characters "inf" (infinity) will be placed in the string (the fourth argument is set to indicate positive or negative infinity). The string itself contains neither a minus sign nor a decimal point nor a base ten exponent.

DIAGNOSTICS

SEE ALSO

ecvt(), itoa()

NOTES

NAME

fgets - read file into string

SYNOPSIS

```
#include "stdio.h"
int     fgets (s,n,fp)
char   *S;
int    n;
FILE   *fp;
```

DESCRIPTION

Fgets will read a line of up to n characters from the file pointed to by its third argument into the area pointed to by its first argument. Its third argument must be a file pointer previously returned by an fopen call. Fgets returns a pointer to the start of the line read or NULL if for some reason no line could be read. The function reads the number of characters indicated by its second argument or until an end of line is encountered, whichever comes first. The trailing newline IS included in the line read.

DIAGNOSTICS

fgets will return NULL if the file could not be read from; this is usually interpreted as an End Of File.

SEE ALSO

fclose(), fflush(), fopen(), fprintf(), fputs(), fscanf(), getc(), gets()

NOTES

If there is a trailing newline character read from the file fgets will include it in the string whereas gets will not.

NAME
   _filespec - Build file specification

SYNOPSIS

```
*include "stdio.h"
int     _filespec(n,fp,ext)
char    *n;
FILE    *fp;
char    ext;
```

DESCRIPTION

The _filespec function builds a file specification in the fcp pointed to by the second argument. The first argument points to a file name string that may contain a drive specifier and an extension. If no drive is given in the name, the system working disk is assumed. If no extension is given in the name, the value of the third argument is used in a call to the FLEX routine SETEXT to set the default extension. (see "The FLEX Advanced Programmers Guide" for more details on the ext parameter.)

DIAGNOSTICS

Returns ERROR if a valid file specification could not be made.

SEE ALSO

NOTES

This routine is used internally by some of the file routines and is not guaranteed to be supported in the future.

NAME

_fms - Call to FLEX FMS entry point

SYNOPSIS

```
#include "stdio.h"
int     _fms(fp,c)
FILE    *fp;
char    C;
```

DESCRIPTION

This is a short assembly language routine that allows a C program to call the FLEX FMS entry point. The desired function should be placed in fp->f.function (see the flex.h header file). The value of the second parameter is placed in the A accumulator before the call to the FMS entry point. On return, fms returns an integer representing the value of the A Accumulator or ERROR.

DIAGNOSTICS

Returns ERROR if FLEX detected an error in the FMS call.

SEE ALSO

NOTES

This routine is used internally by some of the file routines and is not guaranteed to be supported in the future.

NAME

fopen - open a file

SYNOPSIS

```
#include        "stdio.h"
FILE    *fopen(name,mode)
char    *name,*mode
```

DESCRIPTION

Fopen will open the file whose name is pointed to by its first argument with the attributes indicated in the string pointed to by its second argument. It returns a value of type pointer to FILE which must be used as an argument on subsequent references to the file.

The options with which the file is to be opened are specified as ASCII characters in the mode string (whose pointer is passed as the second parameter). One of the characters in this string indicates the mode for which the file will be opened. The appropriate modes are:

r - read: File is opened for read access

w - write: File is opened for write access

If neither of these characters appears in the string the file is opened for read access. The result of placing more than one of these characters in the string is undefined.

In addition to one of the preceding characters a b may appear in the string. The 'b' option indicates that the file is a binary file while the absence of a 'b' indicates that the file should be opened as a text file.

DIAGNOSTICS

Fopen will return ERROR if the file could not be opened and the external variable "errno" will contain any error code returned by the system.

SEE ALSO

fclose(), fgets(), fprintf(), fputs(), fscanf(), getc()

NOTES

The current version of fopen returns ERROR when it fails to open a file rather than the more common return value of NULL.

NAME

fprintf - formatted output conversion

SYNOPSIS

```
#include      "stdio.h"
int      fprintf(stream,control [,arg])
FILE   *stream;
char   *control;
```

DESCRIPTION

Fprintf is nearly identical to printf except that here the output file specification is explicitly given as the first argument. All output is sent to the file pointed to by the first argument. The parameters to fprintf consist of pointer to FILE, followed by a pointer to a null terminated string, followed by zero or more arguments. fprintf formats and writes the arguments following the control string using the control string to direct formatting and conversion. The control string may contain normal characters (which are simply copied to the output file) and conversion specifications which control the writing of the arguments. Each conversion provides information used to format its corresponding argument following the control string. Conversion specifications begin with a percent character (%), perhaps followed by some options and terminated by a conversion character. All the options are, of course, optional but those that are included must appear in the specified order. The legal options (in the order they must appear) are as follows:

Dash (-): indicates that if the number to be written is shorter than the specified field length that it should be left justified. If this option is omitted the number will be right justified.

Zero (0): indicates that if the number to be written is shorter than the specified field length that it should be padded with zeros to fill the field length. If this option is omitted the field will be padded with blanks.

Digit string: indicates the minimum field width. The argument will be written in a field at least this wide. This field may be replaced with a star (*) which will cause the field width to be taken from the next corresponding argument (of type integer) in the argument list.

Period (.): separates the field width from the next digit string.

Digit string: indicates the precision. For a float the precision is the number of digits to be written to the right of the decimal point. For a string the precision is the maximum number of characters which will be written. This field may be replaced with a star (*) which will cause the field width to be taken from the next corresponding argument (assumed to be an integer) in the argument list

Long (l): (letter ell) indicates that the corresponding argument is to be written as a long rather than an int.

The valid conversion characters and their meanings are as follows:

d The argument is assumed to be of type int and is written in decimal notation.

o The argument is written in octal (without leading 0).

x Argument is written in hexadecimal (without leading Ox).

u The argument is assumed to be unsigned and written in decimal notation.

c The argument is written as a character.

s The argument is assumed to be a pointer to a null terminated string. Characters are copied from the control string to the output string until a null character is reached or until the number of characters given by the precision are copied. The terminating null is not copied.

e The argument is assumed to be a float and written out in a decimal notation of the following form: [-d.dddddde[+|-]dd That is a negative sign if the number is negative, a single digit, followed by a decimal point, followed by several digits, followed by an 'e', followed by a sign, followed by two digits.

f The argument is assumed to be a float and written out in a decimal notation of the following form: [-]ddd.dddd where the length of the string of digits following the decimal point is given by the precision.

g Prints in either e or f format; whichever is shorter.

If a character which is neither an option nor a conversion character is found while scanning a conversion specification the character following the percent sign (%) is simoly written and no conversion specification is assumed. Thus to write a percent sign one writes it twice(%%).

DIAGNOSTICS

Fprintf returns ERROR if it fails.

SEE ALSO

printf(),sprintf()

```
NAME
     fputs - write a string to a file

SYNOPSIS
     #include  "stdio.h"
     int      fputs(s,fp)
     char     *S;
     FILE     *fp;

DESCRIPTION
     Fputs copies the string pointed to by the first argument to the
     file  indicated  by the second   argument.  The second argument
     of type pointer to FILE and should have been returned by a call
     to fopen unless it is STDOUT or STDERR.

DIAGNOSTICS
     Returns  ERROR  if an error occurred  while attempting to write
     the string.

SEE also
       puts()

NOTES
```

NAME
    free - free memory

SYNOPSIS
```
     char    *free(block)
     char    *block;
```

DESCRIPTION
    Free will attempt to free a block of memory indicated by its argument. The only valid argument for free is a pointer previously returned by an alloc call. This routine should only be used to free a block that has been allocated via alloc. The result of freeing the same block of memory more than once or attempting to use, as an argument, a pointer which was not returned by an alloc call is undefined (bad things happen).

DIAGNOSTICS

SEE ALSO
    alloc(), sbrk()

NOTES

NAME

fscanf - formatted input conversion

SYNOPSIS

```
#include      "stdio.h"
int     fscanf(file,control [,pointer1]...)
FILE    *file;
char    *control;
```

DESCRIPTION

Fscanf is nearly identical to scanf except that the input file specification is explicitly stated; the input is taken from the file pointed to by the first argument. The parameters to fscanf consist of a pointer to file, followed by a pointer to a null terminated string (the control string), followed by zero or more arguments of type pointer. Fscanf reads groups of characters from the input file pointed to by the first argument, interprets them according to the control string, and writes the results into the arguments pointed to by their corresponding argument pointers. The control string may contain blanks, tabs, and newlines which match optional white space in the input; it may contain ordinary characters which must match the input string exactly character per character; and it may contain conversion specifications used to control the interpretation of the input stream. Each conversion specification provides information used to translate a segment of the input stream into a value which may then be placed into an argument pointed to by its corresponding pointer in the argument list.

Conversion specifications begin with a percent character perhaps followed by some options, and terminated by a conversion character. All the options are, of course, optional but those that are included must appear in the specified order. The legal options (in the order they must appear) are:

Star (*): indicates that this conversion specification has no corresponding pointer in the argument list. This effectively skips a value in the input stream.

Digit string: indicates the maximum field width; the maximum number of characters which this conversion specification will cause to be read from the input stream.

Long (l): (letter ell) indicates that the corresponding pointer is pointing to a long rather than an int. This has no effect when preceding an e or f.

The valid conversion characters and their meanings are as follows:

d   A decimal integer is expected in the input string. Its corresponding pointer is assumed to be of type *int.

o   An octal integer is expected in the input string. Its corresponding pointer is assumed to be of type *int.

x   A hexadecimal integer is expected in the input string. Its corresponding pointer is assumed to be of type lint.

h   A decimal integer is expected in the input string. Its corresponding pointer is assumed to be of type short.

u   An unsigned decimal integer is expected in the input string. Its corresponding pointer is assumed to be of type *unsigned.

c   The very next character is read from the input string (even if it's a blank). Its corresponding pointer is assumed to be of type char.

s   A string is expected in the input string. Its corresponding pointer is assumed to be of type *char. It should point to a space large enough to hold the input string plus an added null. Characters are read, starting with the next nonblank character, until the number of characters given in the precision is reached or until a blank, tab, or newline is reached.

e   (same as f)

f   A floating point number is expected in the input string. Its corresponding pointer is assumed to be of type *float.

DIAGNOSTICS

The return value of this function is the number of parameters that were matched (read in from the input line) or EOF (-1).

SEE ALSO

scanf(), sscanf()

NOTES

Exactly one line of input is consumed for each call to fscanf. Thus fscanf will not fetch a new line even though there are still conversion specifications left to process nor will it save any input left from the preceding line for the next call to fscanf.

A hexadecimal number may not be preceded by a 0x.

Any character within a conversion specifier which is not a legal conversion specifier option or conversion character will be ignored along with the preceding percent sign and any characters inbetween. Thus there is no way to match a '%' on the input line.

NAME
     getc - get the next character from a file

SYNOPSIS
```
     #include "stdio.h"
     int     getc(fp)
     FILE    fp;
```

DESCRIPTION
     Getc returns the next character from the file indicated by its argument.   Its argument is of type pointer to FILE and should have been previously returned from an fopen call unless it is STDIN.

DIAGNOSTICS
     Getc returns ECF (-1) upon reading end of file or on error.

SEE ALSO
     getchar()

NOTES
     Notice the return value of getc is an integer not a character. This is so that getc can return ECF (-1) on end of file.

NAME
     getchar - get a character from the standard input

SYNOPSIS
```
     int     getchar()
```

DESCRIPTION
     Getchar is identical to getc(stdin).   It returns the next character from the standard input.

DIAGNOSTICS
     Getchar returns ECF (-1) upon reading end of file or on error.

SEE ALSO
     getc()

NOTES
     Notice the return value of getchar is an integer not a character.   This is so that getchar can return an ECF (-1) on end of file.

NAME
     _getchr - Call FLEX GETCHR entry point.

SYNOPSIS
```
     #include "stdio.h"
     int      _getchr()
```

DESCRIPTION
     This function returns the value obtained by a call to the FLEX entry point GETCHR (get console character).

DIAGNOSTICS

SEE ALSO

NOTES
     This routine is used internally by some of the file routines and is not guaranteed to be supported in the future.

NAME

gets - read input into string

SYNOPSIS

```
int        gets(s)
char       *S;
```

DESCRIPTION

Gets will read a line from the standard input (STDIN) into the area pointed to by its argument. Gets returns a pointer to the start of the line read, or NULL if for some reason no line could be read. The function reads until an end of line is encountered. The trailing newline is NOT included in the line read (compare this with fgets(s,n,stdin)).

DIAGNOSTICS

Gets will return NULL on end of file and error.,

SEE ALSO

Fclose(), fflush(), fgets(), fopen(), fprintf(), fputs(), fscanf(), getc().

NOTES

Gets will not include any trailing newline character in the string whereas fgets will.

NAME
     index - find first occurrence of character

SYNOPSIS
```
     int     index(s,c)
     char    *s;
     char c;
```

DESCRIPTION
     Index searches the string whose pointer is passed as its first argument and returns a pointer to the first occurrence of the character specified by the second argument. A zero is returned if the character does not appear in the string.

DIAGNOSTICS

SEE ALSO
     rindex()

NOTES

NAME
isalpha - test for alpha character

SYNOPSIS
```
int     isalpha(ch)
char    ch;
```

DESCRIPTION
Returns true (non zero) if its argument is an alpha character (a through z or A through Z); otherwise returns false (zero).

DIAGNOSTICS

SEE ALSO
isdigit(), islower(), isspace(), isupper()

NOTES

NAME

isdigit - test for digit

SYNOPSIS

```
int     isdigit(ch)
char    ch;
```

DESCRIPTION

Returns true (non zero) if its argument is a digit (0 through 9); otherwise returns false (zero).

DIAGNOSTICS

SEE ALSO

isalpha(), islower(), isspace(), isupper()

NOTES

NAME
    islower - test for lower case

SYNOPSIS
```
     int     islower(ch)
     char    ch;
```

DESCRIPTION
    Returns true (non zero) if its argument is a lower case alpha character (a through z); otherwise returns false (zero).

DIAGNOSTICS

SEE ALSO
    isalpha(), isdigit(), isspace(), isupper()

NOTES

```
NAME
     isspace - test for white space

SYNOPSIS
     int     isspace(ch)
     char    ch;

DESCRIPTION
     Returns  true (non zero)  if its argument is  a space, tab, or
     newline character; otherwise returns false (zero).

DIAGNOSTICS

SEE ALSO
     isalpha(), isdigit(), islower(), isupper()

NOTES
```

NAME

isupper - test for upper case

SYNOPSIS

```
int     isupper(ch)
char    ch;
```

DESCRIPTION

Returns true (non zero) if its argument is an upper case alpha character (A through Z); otherwise returns false (zero).

DIAGNOSTICS

SEE ALSO

isalpha(), isdigit(), islower(), isspace()

NOTES

NAME
     itoa - convert integer to ascii string

SYNOPSIS

```
     int     itoa(n,s)
     int     n;
     char    *S;
```

DESCRIPTION
     Itoa converts its first argument into a null terminated ascii string which is stored at the location pointed to by its second argument.   If the integer is negative the string will be preceded by a minus sign. The second argument should point to an area large enough to contain the resultant string which may contain a sign, up to 5 digits, and a NULL termination character.

DIAGNOSTICS

SEE ALSO
     fcvt(), ecvt()

NOTES

NAME

longjmp - non-local goto

SYNOPSIS

```
#include     "stdio.h"
int     longjmp(envp,n)
struct jmp_buf *envp;
int     n;
```

DESCRIPTION

Longjmp works in conjunction with setjmp to provide the ability to jump outside of a function. Compare this to a normal goto for which the destination must be in the same function as the goto statement. Setjmp is used to mark a location as a destination (that is save a copy of the current environment) for later use by the longjmp routine. The argument to setjmp is a pointer to structure which will hold the current environment. A pointer to this structure is used as an argument to longjmp. Longjmp simply restores the environment which was saved by the setjmp call. The effect is that execution continues at the location where the environment was saved (inside the setjmp call). The appearance is that of a return from setjmp.

To mark a location one makes a call to setjmp. This will initialize the contents of the structure whose pointer was passed as an argument. From this call, setjmp will return the value 0. Later, when control is returned here from a longjmp, the return value will be decided by the second argument of the longjmp call.

Now a jump can be made to this location by making a call to longjmp, using a pointer to the same structure that was initialized by setjmp as the first argument and an integer as the second argument. The second argument, will be used as the return value when control is transferred to the setjmp environment

The destination of a longjump must be in a function which has not itself returned inbetween the call to setjmp and the call to longjmp. That is, the destination of a longjmp must be within a currently active function.

DIAGNOSTICS

SEE ALSO

NOTES

```
NAME
     malloc   allocate memory

SYNOPSIS
     char    *malloc(size)
     int     size;

DESCRIPTION
     malloc will attempt  to allocate a block of memory whose size is
     given by the argument.  If it is successful it returns a pointer
     to that memory, otherwise it returns NULL.

DIAGNOSTICS
     Returns NULL if the memory could not be allocated.

SEE ALSO
     free(), sbrk()

NOTES
```

NAME
     max - return the maximum of two values

SYNOPSIS
```
     int     max(a,b)
     int     a,b;
```

DESCRIPTION
     Max returns the greater of its two arguments.

DIAGNOSTICS

SEE ALSO
     min()

NOTES

NAME

min - return the minimum of two values

SYNOPSIS

```
int     min(a,b)
int     a,b;
```

DESCRIPTION

Min returns the lesser of its two arguments.

DIAGNOSTICS

SEE ALSO

max()

NOTES

NAME
     modf - return fractional part of float

SYNOPSIS
```
     float   modf(fp,fint)
     float   fp;
     float   *fint;
```

DESCRIPTION
     Modf takes a floating point number as its first argument and returns its fractional part. Its nonfractional part is written to the location pointed to by the second argument.

     This routine is used by ecvt and fcvt.

DIAGNCSTICS

SEE ALSO

NOTES

NAME

movmem - copy a block of memory from one location to another

SYNOPSIS

```
int      movmem (from,to,length)
char     *from, *to;
unsigned         length;
```

DESCRIPTION

Movmem copies the number of bytes given by the third argument from the location pointed to by first argument to the location pointed to by the second argument. The new copy will exactly reflect the original as it existed before the call even if the two blocks of memory overlap (in that case, of course, the original will be partially overwritten).

DIAGNOSTICS

SEE ALSO

NOTES

NAME

printf - formatted output conversion

SYNOPSIS

```
int     printf(control [,arg]...)
char    *control;
```

DESCRIPTION

Printf is nearly identical to fprintf excect that there is no output file specification explicitly stated; the result is written to stdout. The parameters to printf consist of a pointer to a null terminated string followed by zero or more arguments. Printf formats and writes the arguments following the control string using the control string to direct formatting and conversion. The control string may contain normal characters (which are simply copied to the output file) and conversion specifications which control the writing of the arguments. Each conversion specification provides information used to format its corresponding argument following the control string. Conversion specifications begin with a percent character (%), perhaps followed by some options and terminated by a conversion character. All the options are, of course, optional but those that are included must appear in the specified order. The legal options (in the order they must appear) are as follows:

Dash (-): indicates that if the number to be written is shorter than the specified field length, it should be left justified. if this option is omitted the number will be right justified.

Zero (0): indicates that if the number to be written is shorter than the specified field length, it should be padded with zeros to fill the field length. If this option is omitted the field will be padded with blanks.

Digit string: indicates the minimum field width. The argument will be written in a field at least this wide. This field may be replaced with a star (*) which will cause the field width to be taken from the next corresponding argument (assumed to be an integer) in the argument list.

Period (.): separates the field width from the next digit string.

Digit string: indicates the precision. For a float the precision is the number of digits to be written to the right of the decimal point. For a string the precision is the maximum number of characters which will be written. This field may be replaced with a star (*) which will cause the field width to be taken from the next corresponding argument (assumed to be an integer) in the argument list.

Long (l): (letter ell) indicates that the corresponding argument is to be written as a long rather than an int.

The valid conversion characters and their meanings are as follows:

d The argument is assumed to be of type int and is written in decimal notation.

o The argument is written in octal (without leading 0).

x Argument is written in hexadecimal (without leading Ox).

u The argument is assumed to be unsigned and written in decimal notation.

c The argument is written as a character.

s The argument is assumed to be a pointer to a null terminated string. Characters are copied from the control string to the output string until a null character is reached or until the number of characters given by the precision are copied. The terminating null is not copied.

e The argument is assumed to be a float and written out in a decimal notation of the following form: [-]d.dddddde[+|-]dd That is a negative sign if the number is negative, a single digit, followed by a decimal point, followed by several digits, followed by an 'e', followed by a sign, followed by two digits.

f The argument is assumed to be a float and written out in a decimal notation of the following form: [-]ddd.dddd where the length of the string of digits following the decimal point is given by the precision.

g Prints in either e or f format; whichever is shorter.

If a character which is neither an option nor a conversion character is found while scanning a conversion specification the character following the percent sign (%) is simply written and no conversion specification is assumed. Thus to print out a percent sign one writes it twice (%%). A space is NOT a legal option.

DIAGNOSTICS

Printf returns ERROR if it fails.

SEE ALSO

fprintf(), sprintf()

NOTES

NAME
putc - write a character to a file

SYNOPSIS
```
#include  "stdio.h"
int      putc(c,fp)
char     c;
FILE     *fp;
```

DESCRIPTION
Putc sends the character given as its first argument to the file whose file pointer is given as its second argument. The file pointer must have been previously returned from an fopen call unless it is STDOUT or STDERR.

DIAGNOSTICS
Putc returns ERROR (-1) if an error occurs during the write process.

SEE ALSO

NOTES

NAME

putchar - write a character to the standard output

SYNOPSIS

```
int     putchar(c)
char    C;
```

DESCRIPTION

Putchar sends the character given as its argument to STDOUT. A call of the form putchar(c) is identical to putc(c,stdout).

DIAGNOSTICS

Putchar returns ERROR (-1) if an error occurs during the write process.

SEE ALSO

putc()

NOTES

NAME
     putchr - Call FLEX PUTCHR entry point.

SYNOPSIS
```
     #include "Istdio.h"
     int      _putchr(c)
     char     c;
```

DESCRIPTION
     This function performs a call to the FLEX entry point PUTCHR to perform console output.

DIAGNOSTICS

SEE ALSO

NOTES
     This routine is used internally by some of the file routines and is not guaranteed to be supported in the future.

NAME
puterr - write a char to the standard error output (STDERR)

SYNOPSIS
```
int     puterr(c)
char    c;
```

DESCRIPTION
Puterr sends the character given as its argument to STDERR. A call of the form puterr(c) is identical to putc(c,stderr).

DIAGNOSTICS
Puterr returns ERROR (-1) if an error occurs during the write process.

SEE ALSO

NOTES
STDERR is always directed to the terminal.

NAME
puts - write a string to standard output

SYNOPSIS
```
int    puts(s)
char   *s;
```

DESCRIPTION
Puts copies the string pointed to by the argument to the standard output. The effect is the same as fputs(s,stdout).

DIAGNOSTICS
Returns ERROR if an error occurred while attempting to write the string.

SEE ALSO
fputs()

NOTES
Does NOT append a newline (contrary to some implementations).

NAME

reverse - reverse a string in place

SYNOPSIS

```
int     reverse(s)
char    *s;
```

DESCRIPTION

Reverses the order of the elements of a string pointed to by the argument. If the string the argument pointed to was "abcdef" before the call, it would be "fedcba" after the call.

DIAGNOSTICS

SEE ALSO

NOTES

NAME

rewind - reset specified file to beginning

SYNOPSIS

```
#include "stdio.h"
int      rewind(fp)
FILE     *fp;
```

DESCRIPTION

Rewind resets the file back to the beginning.

DIAGNOSTICS

Returns ERROR for improper file specification.

SEE ALSO

NOTES

NAME

rindex - find last occurrence of character

SYNOPSIS

```
int     rindex(s,c)
char    *S;
char c;
```

DESCRIPTION

Rindex searches the string whose pointer is passed as its first argument and returns a pointer to the last occurrence of the character specified by the second argument. A zero is returned if the character does not appear in the string.

DIAGNOSTICS

SEE ALSO

index()

NOTES

NAME
     sbrk - allocate memory

SYNOPSIS

```
     char     *sbrk(size)
     int      size;
```

DESCRIPTION
     Sbrk will attempt to allocate a block of memory whose size is given by the argument. If it is successful it returns a pointer to that memory; otherwise it returns ERROR.

     Sbrk is similar to alloc except that there is no way to return the memory to the system.

DIAGNOSTICS
     Returns ERROR (-1) if the memory could not be allocated.

SEE ALSO
     alloc(), brk(), free()

NOTES

NAME
scanf - formatted input conversion

SYNOPSIS
```
int    scanf(control [,pointer1] ... )
char   *control;
```

DESCRIPTION
Scanf is nearly identical to fscanf except that there is no input file specification explicitly stated; the input is taken from stdin. The parameters to scanf consist of a pointer to a null terminated string (the control string) followed by zero or more arguments of type pointer. Scanf reads groups of characters from the standard input, interprets them according to the control string and writes the results into the arguments pointed to by their corresponding argument pointers. The control string may contain blanks, tabs, and newlines which match optional white space in the input; it may contain ordinary characters which must match the input string exactly character per character; and it may contain conversion specifications used to control the interpretation of the input stream. Each conversion specification provides information used to translate a segment of the input stream into a value which may then be placed into an argument pointed to by its corresponding pointer in the argument list. Conversion specifications begin with a percent character (%), perhaps followed by some options, and terminated by a conversion character. All the options are, of course, optional but those that are included must appear in the specified order.

The legal options (in the order they must appear) are as follows:

Star (*): indicates that this conversion specification has no corresponding pointer in the argument list. This effectively skips a value in the input stream.

Digit string: indicates the maximum field width; the maximum number of characters which this conversion specification will cause to be read off the input stream.

Long (letter ell) indicates that the corresponding pointer is pointing to a long rather than an int. This has no effect when preceding an e or f.

The valid conversion characters and their meanings are as follows:

d	A decimal integer is expected in the input string. Its corresponding pointer is assumed to be of type *int.
o	An octal integer is expected in the input string. Its corresponding pointer is assumed to be of type *int.

x A hexadecimal integer is expected in the inout string. Its corresponding pointer is assumed to be of type lint.

h A decimal integer is expected in the input string. Its corresponding pointer is assumed to be of type short.

u An unsigned integer is expected in the input string. Its corresponding pointer is assumed to be of type *unsigned.

c The very next character is read from the input string (even if it's a blank). Its corresponding pointer is assumed to be of type *char.

s A string is expected in the input string. Its corresponding pointer is assumed to be of type *char. It should point to a space large enough to hold the input string plus an added null. Characters are read, starting with the next nonblank character, until the number of characters given in the precision is reached or until a blank, tab, or newline is reached.

e (same as f)

f A floating point number is expected in the input string Its corresponding pointer is assumed to be of type *float.

The return value of this function is the number of parameters that were matched (read in off the input line) or ECF.

DIAGNOSTICS

SEE ALSO

fscanf(), sscanf()

NOTES

Exactly one line of input is consumed for each call to scanf. Thus scanf will not fetch a new line even though there are still conversion specifications left to process nor will it save any input left from the preceding line for the next call to scanf. If, for example, one makes a call to scanf with a control string which indicates 3 arguments are expected while only 2 appear on the input line scanf will NOT continue to read lines. Fscanf will simply return with a value of 2. Likewise if the input line had contained 4 arguments only 3 would have been read while the fourth would be discarded.

A hexadecimal number may not be preceded by a Ox.

Any character within a conversion specifier which is not a legal conversion specifier option or conversion character will be ignored along with the preceding percent sign and any characters in between. Thus there is no way to match a '%' on the input line.

NAME

     _setext - Call FLEX SETEXT entry point

SYNOPSIS

```
     #include "stdio.h"
     int     _setext(fp,ext)
     FILE     fp;
     char    ext;
```

DESCRIPTION

     The _setext function performs a call to the FLEX routine SETEXT to set a default file name extension into the given file control block.

DIAGNOSTICS

SEE ALSO

NOTES

     This routine is used internally by some of the file routines and is not guaranteed to be supported in the future.

NAME

setjmp - non-local goto

SYNOPSIS

```
#include
int     setjmp (envp)
       jmp_buf *envp;
```

DESCRIPTION

Setjmp works in conjunction with longjmp to provide the ability to jump outside of a function. Compare this to a normal goto for which the destination must be in the same function as the goto statement. Setjmp is used to mark a location as a destination (that is save a copy of the current environment) for later use by the longjmp routine. The argument to setjmp is a pointer to structure which will hold the current environment. A pointer to this structure is used as one of the arguments to longjmp. Longjmp simply restores the environment which was saved by the setjmp call. The effect is that execution continues at the location where the environment was saved (inside the setjmp call). The appearance is that of a return from setjmp.

To mark a location one makes a call to setjmp. This will initialize the contents of the structure whose pointer was passed as an argument. From this call setjmp will return the value 0. Later, when control is returned here from a longjmp, the return value will be decided by the second argument of the longjmp call. (see longjmp)

Now a jump can be made to this location by making a call to longjmp using a pointer to the same structure that was initialized by setjmp as the first argument and an integer as the second argument. The second argument will be used as the return value when control is transferred to the setjmp environment.

The destination of a longjmp must be in a function which has not itself returned inbetween the call to setjmp and the call

to longjmp.

DIAGNOSTICS

SEE ALSO

longjmp()

NOTES

NAME
  sprintf - formatted output conversion

SYNOPSIS

```
     int     sprintf(string,control [,arg1]...)
     char    *string, *control;
```

DESCRIPTION

Sprintf is nearly identical to printf except that rather than writing to the standard output (stdout), the result is placed in a null terminated string pointed to by the first argument (which is assumed to be of type pointer to character). The parameters to sprintf consist of a pointer to char, followed by a pointer to a null terminated string, followed by zero or more arguments. Sprintf formats the arguments following the control string, using the control string to direct formatting and conversion. It places the result in the string pointed to by the first argument which must be long enough to accept it. The control string may contain normal characters (which are simply copied to the output string) and conversion specifications which control the cooying of the arguments. Each conversion specification provides information used to format its corresponding argument following the control string. Conversion specifications begin with a percent character, (%), perhaps followed by some options, and terminated by a conversion character. All the options are, of course, optional but those that are included must appear in the specified order. The legal options (in the order they must appear) are as follows:

Dash (-): indicates that, if the number to be copied is shorter than the specified field length, it should be left justified. if this option is omitted the number will be right justified.

Zero (0): indicates that, if the number to be copied is shorter than the specified field length, it should be padded with zeros to fill th field length. If this option is omitted the field will be padded with blanks.

Digit string: indicates the minimum field width. The argument will be copied into a field at least this wide. This field may be replaced with a star (*) which will cause the field width to be taken from the next corresponding argument (assumed an integer) in the argument list.

Period (.): separates the field width from the next digit string.

Digit string: indicates the precision. For a float the precision is the number of digits to be written to the right of the decimal point. For a string the precision is the maximum number of characters which will be written. This field may be replaced with a star (*) which will cause the field width to be taken from the next corresponding argument (assumed to be an integer) in the argument list

Long (l): (letter ell) indicates that its corresponding argument is to be written as a long rather than an int.

The valid conversion characters and their meanings are as follows:

d The argument is assumed to be of type int and is written in decimal notation.

o The argument is written in octal (without leading 0).

x Argument is written in hexadecimal (without leading Ox).

u The argument is assumed to be unsigned and written in decimal notation.

c The argument is written as a character.

s The argument is assumed to be a pointer to a null terminated string. Characters are copied from the control string to the output string until a null character is reached or until the number of characters given by the precision are copied. The terminating null is not copied.

e The argument is assumed to be a float and written out in a decimal notation of the following form: [-]d.dddddde[+|-]dd That is a negative sign if the number is negative, a single digit, followed by a decimal point, followed by several digits, followed by an 'e', followed by a sign, followed by two digits.

f The argument is assumed to be a float and written out in a decimal notation of the following form: [-]ddd.dddd where the length of the string of digits following the decimal point is given by the precision.

g Prints in either e or f format; whichever is shorter.

if a character which is neither an option nor a conversion character is found while scanning a conversion specification the character following the percent sign (%) is simply written and no conversion specification is assumed. Thus to write a percent sign one writes it twice (%%)

DIAGNOSTICS

SEE ALSO
printf(), fprintf()

NOTES

NAME

sscanf - formatted string conversion

SYNOPSIS

```
int     sscanf(string,control [,pointer1] ... )
char    *string, *control;
```

DESCRIPTION

Sscanf is nearly identical to fscanf except that its input is taken from the string pointed to by the first argument rather than a file. The parameters to sscanf consist of a pointer to char, followed by a pointer to a null terminated string (the control string), followed by zero or more arguments of type pointer. Sscanf reads groups of characters from the input string pointed to by the first argument, interprets them according to the control string, and writes the results into the arguments pointed to by their corresponding argument pointers. The control string may contain blanks, tabs, and newlines which match optional white space in the input string; it may contain ordinary characters which must match the input string exactly character per character; and it may contain conversion specifications used to control the interpretation of the input string. Each conversion specification provides information used to translate a segment of the input string into a value which may then be placed into an argument pointed to by its corresponding pointer in the argument list.

Conversion specifications begin with a percent character, (%), perhaps followed by some options, and terminated by a conversion character. All the options are, of course, optional but those that are included must appear in the specified order.

The legal options (in the order they must appear) are as follows:

Star (*) indicates that this conversion specification has no corresponding pointer in the argument list. This effectively skips a value in the input string.

Digit string: indicates the maximum field width; the maximum number of characters which this conversion specification will cause to be read off the input string.

Long (l): (letter ell) indicates that the corresponding pointer is pointing to a long rather than an int. This has no effect when preceding an e or f.

The valid conversion characters and their meanings are as follows:

d A decimal integer is expected in the input string. Its corresponding pointer is assumed to be of type lint.

o   An octal integer is expected in the input string. Its corresponding pointer is assumed to be of type *int.

x   A hexadecimal integer is expected in the input string. Its corresponding pointer is assumed to be of type *int.

h   A decimal integer is expected in the input string. Its corresponding pointer is assumed to be of type *short.

u   An unsigned decimal integer is expected in the input string. Its corresponding pointer is assumed to be of type *unsigned.

c   The very next character is read from the input string (even if it's a blank). Its corresponding pointer is assumed to be of type *char.

S   A string is expected in the input string. Its corresponding pointer is assumed to be of type *char. It should point to a space large enough to hold the input string plus an added null. Characters are read, starting with the next nonblank character, until the number of characters given in the precision is reached or until a blank, tab, or newline is reached.

e   (same as f)

f   A floating point number is expected in the input string. Its corresponding pointer is assumed to be of type *float.

The return value of this function is the number of parameters that were matched (read in off the input line) or EOF.

DIAGNOSTICS

SEE ALSO

scanf(), fscanf()

NOTES

A hexadecimal number may not be preceded by a Ox.

Any character within a conversion specifier which is not a legal conversion specifier option or conversion character will be ignored along with the preceding percent sign and any characters inbetween. Thus there is no way to match a '%' on the input line (i.e. writings %% in the control string will not cause it to try to match a % in the input string).

NAME
     strcat - copy string

SYNOPSIS
```
     int     strcat(sl,s2)
     char    *sl,*s2;
```

DESCRIPTION
     Strcat appends a copy of the string pointed to by its second argument to the end of the string pointed to by its first argument. It is assumed that the first argument points to an area large enough to accomodate the resultant string.

DIAGNOSTICS

SEE ALSO
     strcmp(), strlen(), strsave()

NOTES

```
NAME
     strcmp - compare strings lexicographically

SYNOPSIS
     int     strcmp(sl,s2)
     char    *sl,*s2;

DESCRIPTION
     Strcmp  lexicographically compares its  first argument with its
     second. It returns 1 if the first is greater than the second, 0
     if  the two  are equal, and  -1 if  the first is  less than the
     second.

DIAGNOSTICS

SEE ALSO
     strcpy(), strlen(), strsave()

NOTES
```

```
NAME
     strcpy - copy string

SYNOPSIS
     int     strcpy(sl,s2)
     char    *sl,*s2;

DESCRIPTION
     Strcpy  copies the string pointed to  by the second argument to
     the  area  pointed  to by  the  first.  It stops  after  a null
     character has been conied.

DIAGNOSTICS

SEE ALSO
     strcmp(), strlen(), strsaveo

NOTES
```

```
NAME
     strlen - return string length

SYNOPSIS
     int     strlen(s)
     char    *s;

DESCRIPTION
     Strlen  returns  the length  of the  string  pointed to  by the
     argument (not including the terminating null).

DIAGNOSTICS

SEE ALSO
     strcmp(), stcpy(), strsave()

NOTES
```

NAME

strncat - copy string

SYNOPSIS

```
int      strncat(sl,s2,n)
char     *sl,*s2;
int      n;
```

DESCRIPTION

Strncat appends a copy of the string pointed to by its second argument to the end of the string pointed to by its first argument. Strncat copies at most the number of characters specified by its third argument. It is assumed that the first argument points to an area large enough to accomodate the resultant string.

DIAGNOSTICS

SEE ALSO

strcat(), strcmd(), strlen(), strsave()

NOTES

NAME
     strncmp - compare strings lexicographically

SYNOPSIS
```
     int      strncmp(sl,s2,n)
     char     *sl,*s2;
     int      n;
```

DESCRIPTION
     Strncmp lexicographically compares its first argument with its second. It returns 1 if the first is greater than the second, 0 if the two are equal, and -1 if the first is less than the second. Strncmp compares at most the number of characters specified by its third argument; any others are not considered.

DIAGNOSTICS

SEE ALSO
     strcmp(), strcpy(), strlen(), strsave()

NOTES

NAME
     strncpy - copy string

SYNOPSIS

```
     int     strncpy (s1,s2,n)
     char    *sl,*s2;
     int n;
```

DESCRIPTION
     Strncpy copies the string pointed to by the second argument to the area pointed to by the first. It stops after it has copied the number of characters specified by its third argument or when a null character has been copied.

DIAGNOSTICS

SEE ALSO
     strcmp(), strcpy(), strlen(), strsave()

NOTES

NAME

strsave - save string in memory

SYNOPSIS

```
char    *strsave(s)
char    *S;
```

DESCRIPTION

Strsave attempts to allocate a space in memory large enough to hold the string pointed to by the argument (plus its terminating null). If it succeeds strsave copies the string pointed to by the argument into the memory and returns a pointer to it. If it fails to allocate sufficient memory, strsave returns NULL.

The area used by "strsave" to save the string is obtained by a call to "alloc" and thus may be returned to the system by a call to "free" using the string pointer as an argument.

DIAGNOSTICS

SEE ALSO

alloc(), free(), strcmp(), strcpy(), strlen()

NOTES

NAME
     tolower - convert to lower case

SYNOPSIS

```
     char    tolower(ch)
     char    ch;
```

DESCRIPTION
     Returns its argument converted to lower case

DIAGNOSTICS

SEE ALSO
     toupper()

NOTES

NAME
     toupper - convert to upper case

SYNOPSIS
```
     char    toupper(ch)
     char    ch;
```

DESCRIPTION
     Returns its argument converted to upper case

DIAGNOSTICS

SEE ALSO
     tolower()

NOTES

NAME

uldiv   unsigned long integer divide

SYNOPSIS

```
long    uldiv(opl,op2)
long    opl,op2;
```

DESCRIPTION

Uldiv returns a long (unsigned) integer which represents the nonfractional result of dividing the first (unsigned) long integer argument by the second (unsigned) long integer argument.

DIAGNOSTICS

Division by 0 will return (long) -1.

SEE ALSO

ulmod(), ulmul()

NOTES

There is actually no type "unsigned long". Uldiv operates on longs as if they were unsigned by ignoring the normal sign conventions.

NAME
     ulmod - unsigned long modulo operation

SYNOPSIS

```
     long    ulmod (opl, op2)
     long    opl,op2;
```

DESCRIPTION
     Ulmod returns a long (unsigned) integer which represents the remainder of the result produced by dividing the first (unsigned) long integer argument by the second (unsigned) long integer argument.

DIAGNOSTICS
     When the second argument is zero (division by 0) the function returns the first argument.

SEE ALSO
     uldiv(), ulmul()

NOTES
     There is actually no type "unsigned long". Ulmod operates on longs as if they were unsigned by ignoring the normal sign conventions.

NAME
ulmul - unsigned long multiply

SYNOPSIS
```
long    ulmul (opl, op2)
long    opl,op2;
```

DESCRIPTION
Ulmul returns a long (unsigned) integer which represents the result of multiplying the first (unsigned) long integer argument by the second (unsigned) long integer argument.

DIAGNOSTICS

SEE ALSO
uldiv(), ulmod

NOTES
There is actually no type "unsigned long". Ulmul operates on longs as if they were unsigned by ignoring the normal sign conventions.

NAME

_unext - unextend float

SYNOPSIS

```
     float    unext(ef)
     struct   extflt
              {
              char    sign;
              int     exp;
              long    mantissa;
              } *ef;
```

DESCRIPTION

_unext returns the float which is represented by the extended floating point number contained in the structure pointed to by the argument. The first element of the structure is assumed to contain the sign bit of the number, the second element should contain the unbiased exponent, and the third the mantissa.

DIAGNOSTICS

SEE ALSO

_extend()

NOTES

NAME

ungetc - push character back on input stream

SYNOPSIS

```
#include "stdio.h"
int     ungetc (c, fp)
FILE    *fp;
int     c;
```

DESCRIPTION

Ungetc attempts to push a character back on the input stream so that it will be the next one retrieved. At most one character may be pushed back inbetween calls to getc. The first argument is the character to be pushed the second is a pointer to the file into which the character is to be pushed. The file pointer must have been previously returned from an fopen call unless it is STDIN.

DIAGNOSTICS

Ungetc returns ERROR (-1) if it could not push the character.

SEE ALSO

getc()

NOTES

NAME
    ungetchar - push character back on standard innut stream

SYNOPSIS

```
#include "stdio.h"
int      ungetchar(c)
char     c;
```

DESCRIPTION
    Ungetchar attempts to push a character back on the standard input stream so that it will be the next one retrieved. At most one character may be pushed back inbetween calls to getchar. The argument is the character to be pushed. This call is equivalent to ungetc (c, STDIN)

DIAGNOSTICS
    Ungetchar returns ERROR (-1) if it could not push the character.

SEE ALSO

NOTES

NAME
     unlink - delete file

SYNOPSIS
```
     int     unlink(name)
     char    *name;
```

DESCRIPTION
     Unlink deletes the file whose name is contained in the string pointed to by its argument. Under some operating systems unlink simply decreases a link count to the file and deletes the file if the link count reaches zero as a result.

DIAGNOSTICS
     Unlink returns ERROR if the file could not be cveleted.

SEE ALSO

NOTES
     Under the Flex and 0S9 operating systems unlink simply has the effect of deleting the file. Under more Unix like operating systems such as UniFLEX unlink decreases the link count on the file. Such an operating system will delete any file whose link count decreases to zero. There is a companion library routine, link(), which increases the link count on a file for those operating systems which support it.

ADDENDUM TO THE INTROL-C USER MANUAL

LINKER AND LOADER REFERENCE MANUAL

-b Option

Two forms of the' "-b" option described on page L.1.6 of the Linker And Loader Reference Manual are now available:

```
      -b      -or-     -b=<Pathnarne>
```

The first form above, "-b", prevents the Standard Library, libc.R, from being searched by the Linker. The second form, "-b=<Pathname>", defines <pathname> as being a non-standard place in which to find the Standard Library, libc.R.

-i Option

A "-i" option has been added for the Linker. When. a -i is specified on the link command line, this option specifier will force loading of all modules on the command line.

-l Option

Two forms of the "-l" option described on page L.1.8. of the Linker And Loader Reference Manual are now available:

```
    -l[s][x][u][=<file>]         -or-          -ll[s][x][u][=<file>]
```

The first form above, where a single leading "1" is specified, causes a linker listing to be produced exactly as described on page L.1.8 of the User Manual. The second form, where a double leading "l" is used, instead causes a loader listing to be produced. That is, an option specification beginning with "-l" will be ignored by the linker itself and passed intact to the loader to cause a loader listing to be generated.

-r option

A "-r" option has been added for the Linker. The -r option specifier causes the .RL output file generated by the Linker to be saved during an automatic link-and-load sequence. Normally (when the -r option is not specified), when the Linker automatically calls the Loader, the Linker passes the Loader a "-z" option specifier which causes the Loader to delete its input file (ie the Linker's .RL output file) when the Loader has finished with it. Specifying the -r option on the link command line inhibits the Linker from passing the -z specifier to the Loader, thus causing the intermediate RL Linker output file to be retained.

STANDARD LIBRARY REFERENCE MANUAL (UC6809 Library Only)

The Standard Library Reference Manual erroneously describes two routines that do not exist in the supplied Standard Library:

      rand - Return random number
      srand - Set seed for random number generator

Therefore, please delete/ignore the descriptions for these two routines.

APPENDIX A
FC6809 STANDARD LIBRARY

NON-ZERO CLASS LIBRARY ROUTINES

As discussed in the Compiler Reference manual and Linker Reference manual, all relocatable modules (including those contained in the Standard Library) have a special identifying attribute called a "class" specifier, which is a number in the range 0 through 255. At link time, the Linker uses a module's class number to differentiate between different versions of identically named modules that may possibly co-exist within the same library.

In the case of the FC6809 Standard Library, most of the function modules supplied in the library have a preassigned modure class specifier of "O" (zero). In fact, each of the various runtime support functions is furnished and available for use as a class 0 type of module. However, the library also includes "alternate" versions of some runtime functions. Where such alternate support routines exist, they have been given the same filename as the "standard" version of the routine, but have been assigned non-zero class numbers.

In all cases, the class 0 version of a given library routine will always provide the full runtime support features that have been described for that routine in this reference manual. Any non-zero classes of library routines, by comparison, provide a modified (and typically abbreviated) level of support for the given runtime function, usually resulting in smaller runtime overhead in the final program.

Four non-zero class categories of library functions are included in the FC6809 Standard Library; class 5, class 6, class 7, and class 8.

Classes 5 and 6 are associated with selection of modified versions of the output formatting routines, such as printf, fprintf, and sprintf; classes 7 and 8 select modified versions of the input formatting routines, such as scanf, fscanf, and sscanf. Whereas the class 0 versions of these respective routines provide full support for longs, integers, and floating point numbers, the non-zero class versions differ as follows:

<u>Class 5</u> - Output formatting routines will support only integers.

<u>Class 6</u> - Output formatting routines will support only integers and longs.

<u>Class 7</u> - Input formatting routines will support only integers.

<u>Class 8</u> - Input formatting routines will support only integers and longs.

APPENDIX D

INSTALLATION OF THE FC6809 INTROL-C COMPILER

This section describes the installation of Introl-C on the Flex operating system.

The FC6809 Introl-C Compiler is shipped on standard 8 inch or 5 inch floppy disk format. Verify that the disk is indeed intended for the Flex operating system and also that the disk format is what you expect by reading the label on the distribution diskette envelope. Note that the disk shipped to you is not bootable and thus cannot be used to start your Flex system.

Before it can be used, the Compiler and its associated programs must be copied from the distribution disk to the system drive. Unless specified otherwise, the program to be compiled is assumed to be on the work drive.

Notice that the "stdio.h", "flex.h", and "setjmp.h" files are NOT capitalized. When you copy these files, be sure that their names are in lower case. On many FLEX systems file names are automatically converted to upper case even when typed in lower case. Many systems already have a utility to defeat this "feature" but, if not, the distribution disk includes a utivity called "CASE" which, when run, prevents this automatic conversion. The CASE program toggles between 'upper/lower case' and 'upper case only' each time it is run so if it is run an even number of times the system will again convert lower case to upper.

You may also wish to take note of the other files you find on your distribution disk. They include source code examples of many of the standard library routines and perhaps some useful or interesting routines. See your FLEX System Users Manual for details on making copies of files.

INTROL

LINKER AND LOADER
REFERENCE MANUAL

Introl Corp.
647 W. Virginia St.
Milwaukee, WI 53204 USA

tel. (414) 276-2937

Table of Contents

Linker And Loader Reference Manual

LINKER

The function of the Linker is to join several relocatable object modules together to form a single relocatable object module as the result. Normally, when the Introl Linker finishes, it will automatically call the Loader, causing the object module produced by the Linker to be then translated into an executable file by the Loader. Once such executable file has been generated, the actual object module generated by the Linker is normally automatically deleted. Thus, although the Linker itself produces an intermediate relocatable module, the more usual result of a linker command line call is an executable file that is subsequently produced by the Loader. Options are provided, however, to permit the Linker's output module to be retained even though an executable file has been produced; also, an option exists to inhibit the Loader call entirely when the desired result is simply the relocatable module generated by the Linker.

LINKER COMMAND LINE

The general form of the link command line is:

```
ilink <files> {<options>} {<files>} {<options>}
```

where <options> can be zero or more Linker and Loader option specifiers (described later in this Section), and <files> are the filenames of the relocatable files or libraries which are to be input to the Linker. Unless an option to inhibit loading is explicitly specified on the command line (the "-n" option), the Loader will be automatically executed when the Linker finishes.

The Linker expects each of its input files to have a filename extension; if none is explicitly defined, the filename extension is assumed to be ".R", which is the filename extension normally assigned to relocatable files generated by the Assembler. If the Linker is being run independently (ie with the "-n" option specified, which inhibits the automatic call to the Loader), the Linker will produce a relocatable module as the end result, having the filename extension ".RL". Such modules (ie modules which have been linked but not loaded) are themselves relocatable modules which can be legally reused as inputs to the Linker, if desired. If the Loader call is not explicitly inhibited, a link command line call will result in generation of an executable output file as the final result (ie, the file produced by the Loader pass). In this latter case, the intermediate relocatable module generated by the Linker (ie the file having a ".RL" filename extension) will not be retained unless the user specifically opts to do so (via the "-r" Linker option). In either case, the filename assigned to the output module(s) produced as a result of the linker call will be determined by the "primary function name" symbol, which is discussed under Operation, below.

OPERATION

When the Linker is first invoked, it begins its linking process by attempting to resolve two references which are implicit to the Linker. The first is called the "primary function name", the second is the program "entry point". The user may, as an option specification on the link command line (the "-m=<file>" option), specify any symbol as a primary function name. If none is explicitly defined, however, the primary function naming symbol will be assumed to be "_main", the symbol that represents the name of the usual starting function ("main") in a C program. The filename of the module in which the Linker finds the primary function name will normally be the name assigned to the Linker's relocatable output module, but with the filename extension ".RL" being appended to the Linker's output module.

The Linker begins its search by first searching through all of the files specified on the link command line, searching these files in the order they are listed, attempting to resolve the primary function name. If it succeeds, it will include the module which contains the definition of the primary function name, and will then proceed to resolve any external references which that module makes. (If the primary function name cannot be found, the Linker automatically loads the Standard Library and attempts to resolve the "entry point" symbol, as described below.) When all possible external references caused by inclusion of the module containing the primary function name have been satisfied, the Linker will then attempt to resolve the "entry point" symbol. In doing so, the Linker will first search through the files on the link command line, and then search the Standard Library if necessary, looking for a module which has an entry point symbol defined. If it finds one, it will include the module which contains the entry point and attempt to resolve any resultant external references that module makes.

An unmodified Standard Library will always contain a module for which an entry point is defined. This is the module usually used to set up the environment required before the first C function (usually "main") can be executed. The Compiler itself does not normally define an entry point when it produces a module. An assembly language programmer, however, may specify the entry point of an assembly language module by placing the name of the entry point following the END assembler directive. If there is more than one module with an entry point defined, the Linker will assume the entry point is that of the first such module it finds after beginning its search. It begins its search with the files on the link command line, scanning left to right, and then searches the Standard Library, top to bottom. Therefore, if a module on the link command line defines an entry point, that module will be the first module found by the Linker and, therefore, will be the one selected for inclusion (ie rather than the module contained in the Standard Library). If no module on the link command line contains an entry point, the Linker will assume the entry point symbol is "cstart", which happens to be the usual name for the Standard Library routine which sets up the environment for a C program.

The Linker terminates when it has no more external references to resolve or, alternatively, when it runs out of files to search in attempting to satisfy any unresolved references that might still exist. The Linker's output will be a relocatable module that has the same name as the name of the module which contains the primary function name, but with a ".RL" filename extension appended. When the Linker has determined it has resolved all the external references it possibly can, it will automatically call the Loader. If all external references have been successfully resolved by the Linker, the Loader will load the Linker's output into an executable output file. If unresolved references still exist, however, the Loader will complain and loading of the module will be unsuccessful.

As indicated above, it is perfectly legal to use the Linker to link several modules together which, of themselves, do not satisfy all the external references they make. This feature is very useful when it is desired to link two or more relocatable files together to produce a single resultant "partially linked" module (which may contain some unresolved references). Such partially linked modules may themselves then be reused as inputs in subsequent linking operations, and linked with other relocatable modules as necessary. In such-cases, when it is the user's intention to do partial linking of this type, a user option ("-n") to prevent automatic execution of the Loader must be specified on the link command line.

In many cases, such as for a compiled C program contained in a single module, calling the Linker may be as simple as specifing the name of a single relocatable file produced by the Compiler. For example, if the file to be linked and loaded had the name "test.R" (which is the file that would be produced by the Compiler if the user had compiled a program called "test.c"), the user could call the Linker by entering the following:

```
ilink test
```

For this example, the Linker would proceed to first link the file 'test.R" with applicable referenced functions from the Standard Library ("libc.R"), producing the linked module "test.RL" as an intermediate result. It would then automatically call the Loader, which would load "test.RL" into either an executable file or a file of load records, as appropriate to the type of Introl Loader being used. Since the "-r" option was not specified on the linker command line for this particular example, the Loader would also automatically delete the "test.RL" file when it had finished using it. Note that it is unnecessary to specify the Standard Library, "libc.R", on the command line; the Standard Library is always implicit to the Linker when it is called.

LINKER CLASS LIST

Each relocatable module produced by the Assembler, as well as each module contained in the Standard Library, has an attribute called its "class", which is a user-assignable number from "0" (zero) to "255". During the linking process, the Linker always uses the module's class number in combination with the module's filename for module identification purposes. The class number is, in effect, an "extra identifier" that provides a mechanism for distinguishing between several identically named modules that may be contained in a library.

The default "class" for modules produced bv the Assembler is "0"; however, any other legal class number (ie "1" through "255") may be selectively assigned to any of these modules by the user. Similarly, most of the library routines contained in the Standard Library, libc.R, have a preassigned class number of "C", although several non-zero class modules are also supplied. For example, libc.R contains 3 different classes of the ofmt routines used by the "printf", "fprintf", and "sprintf" Standard Library functions (classes 0, 5, and 6) and 3 different classes of the imft routines used by the "scanf", "fscanf", and "sscanf" Standard Library functions (classes 0, 7, and 8). The class 0 ofmt routine supports longs, ints, and floats; the class 5 ofmt routine supports longs only; and the class 6 ofmt routine supports longs and ints. Similarly, the class 0 ifnt routine supports longs, ints, and floats; the class 7 ifmt routine supports only longs; and the class 8 ifmt routine supports longs and ints.

Because of a relocatable module's class attribute, one of the link time options available to the user is the specification of a "linker class list" on the link command line. Use of a class list specification is only necessary when the user wants modules other than class "O" modules to be considered for inclusion by the Linker.

The linker class list specification defines two things to the Linker: (1) it defines the specific non-zero classes of modules that should be potentially considered for that particular link process, and (2) it simultaneously establishes a priority ranking of these classes of modules, which enables the Linker to choose the "correct" module from among possibly several that may have been given identical filenames in a library.

A linker class list is specified on the link command line as one or more <option> entries of the form:

t=<class list>

where <class list> is a series of one or more numerical values from "1" through "255" (see -t option below). The numerical values contained in <class list> represent those specific non-zero module classes, listed in the order in which they are to be "preferred" for possible use, which are to be considered potentially valid for inclusion for that particular link process. Modules of class "O" are ALWAYS implicit in any class list specification and therefore are not included in a linker class list on the command line. The Linker automatically assigns lowest "preference" to class "O" modules and will only use a class 0 module if it cannot find some other identically named module having one of the non-zero classes defined in the linker class list.

As mentioned earlier, a class list specification on the linker command link is only necessary if modules having a class other than "0" are to be considered for use by the Linker. When a class list is specified, however, it is important to note that the order in which any class numbers appear on the command line is just as significant to the Linker as the actual class numbers themselves. This is because the Linker (which scans the entire command line from left to right to determine all of the acceptable classes) assumes that the class numbers are listed by the user in ordered sequence on the command line, with the "most preferred"'class being the class it first encounters on the command line, the "next most preferred" class being the second class it encounters, and so on. The Linker will always select the "most preferred" class of any given named module that it can find.

An ordered class list of this type is necessary for the user to unambiguously define, and the Linker to properly select, the intended module in many instances. For example, suppose the user had compiled and assembled a program module, "file1", (with a class of "0") that referenced two library routines contained in the Standard Library, one called "abc" and the second called "xyz". Further assume that two different versions of the abc module existed, one with class 0 and the other with class 1; and three versions of xyz existed, one with class 0, one with class 1, and one with class 2. If the user wanted to link file1 with the class 1 module of abc and the class 2 module of xyz, he could enter a link command line such as:

```
ilink file1 t=2,1
```

In this case the Linker would ascertain that, given the choice, it should give highest preference to using class 2 modules, next highest preference to class 1 modules, and lowest preference to class 0 modules. During the linking process the Linker would first look for a class 2 filel module and, failing that, then look for a class 1 file1 module and, failing that, then look for a class 0 file 1 module, which it would find and therefore include. The Linker would then begin searching the Standard Library to resolve the references filel makes to abc and xyz. it would begin its search for abc by first looking for an abc class 2 module and, failing that, then begin looking for an abc class 1 module which it would find and link in with filel to resolve the reference made to abc. Similarly, it would begin its search for xyz by first looking for an xyz class 2 module which it will find and link in to file1 to resolve the reference made to xyz. Aithough an abc class 0 module and xyz class 1 and xyz class 0 modules also existed in the library, these would have been ignored by the Linker inasmuch as it had been able to find "more preferred" versions of abc and xyz.

By comparison, if the user had used a link command line such as

```
ilink t=1,2 file1
```

the Linker would instead have given highest preference to class 1 modules and next highest preference to class 2 modules, with class 0 modules again having lowest priority (as is ALWAYS the case for class 0 modules). In this case the Linker would first look for a class 1 file1 module, then a class 2 file1 module, and then a class 0 file1 module which it would find and include. The Linker would then look for, find, and link in the ("most preferred") class 1 abc module; then look for, find, and link in the ("most preferred") class 1 xyz module. The class 2 xyz module would ONLY have been considered for inclusion in this instance if the Linker were unable to find the "more preferred" class 1 module, which of course it does find in the example situation given.

Notice that the class list may contain multiple class specifiers and that class zero is ALWAYS implicit in any class list specification.

LINK COMMAND LINE OPTIONS

Linker options, as well as Loader options, may be specified on the link command line. Loader options, if specified, will be passed on to the Loader when it is automatically called by the Linker. The "linker-specific" options listed below are those options which apply specifically to the Linker, per se. The Loader options that may also be specified on the link command line are discussed in the Loader Appendices to this manual.

Linker-Specific options include:

-b

This option prevents the Standard Library, "libc.R", from being searched by the Linker. Usually this option is specified in combination with the "-f" Linker option, discussed below, when programs are being

-c=<file>

The option specifies that <file> is a command file where the Linker will find additional information. The command file is a text file which may contain extra options and additional file names to be referenced following those listed on the command line. Each option or file name must appear on a separate line in the command file.

-d[<c>]

This option is used for specifying, at link time, which of several (optionally available) Introl Loaders is to be called by the Linker when linking is completed. Specifically, use of this option will cause the Linker to call the Loader whose Introl filename is "<c>ld", where the <c> represents the first character of the desired Loader's "name". For example, the option specification "-dh" would instruct the Linker to call the Loader named "hld" when it finishes (assuming of course that the "hld" Introl Loader is actually available for use). If the -d[<c>] option is not specified, or if there is no character specified via the <c> entry, the Loader selected for use will default to the "standard" Loader supplied with the Compiler. (In general, the "standard" Loader is one which produces code that is executable on the Compiler's host operating system.) The several different types of Loaders that are optionally available for use, and the "<c>ld" names associated with each, are described in the Loader Appendix of this manual.

NOTE: When an "optional" target- system- dependent-type of Loader is being specified for use, the compatible "standard library" supplied with that optional Loader must also be specified for use during the linking process. In such cases the "-b" Linker option can be used to inhibit the Linker's use of the "standard" libc.R library, and the "-f" option used to instruct the Linker to instead find and use the "optional" standard library which is compatible with the target operating system.

-e=<symbol>

This option sets the entry point. If the <symbol> being specifed as the entry point refers to a C symbol that has been generated by the Compiler, the <symbol> name must include a leading underscore character (ie the Compiler automatically pre-pends a leading underscore to all symbols it generates). If this option is not used, the Linker will search through all the modules in the order they are listed on the command line, and then search the Standard Library if necessary, in an attempt to find one which has an entry point defined. The entry point will be that of the first such file the Linker finds. If no input module specifies an entry point, the Linker will usually find one called "cstart" in a module of the same name in the Standard Library. For assembly language programs, an entry point is placed in a module by placing the desired entry point symbol on the "end" directive in an assembly language file (see Assembler section of the Compiler Reference Manual).

-f<string>   or   -f=<string>

This option, which has two forms, is used to specify that additional libraries will be found in the standard library place which are to be searched by the Linker (ie libraries that are to be searched in addition to the Standard Library, libc. R) .The "-f<string>" form of the option specifies that an additional library to be searched is named "lib<string>.R", where <string> represents any series of characters. The "-f=<string>" form specifies that an additional library to be searched is named "<string>.R", where <string> can represent any string of characters. This option must normally be used (together with the "-b" option mentioned above) when an "optional" Loader is being called; this is necessary so that the Linker uses a "standard library" which is compatible with that particular Loader.

-l[s][x][u][=<file>]
This option causes a linker listing to be produced. The optional file name indicates that the listing is to be placed in the indicated file rather than being listed on the console. The "s", "x" and "u" characters are all optional and affect the listing's contents, as follows: If the "s" character is specified the listing will include all symbols. If the "X" character is specified the listing will include a cross reference symbol listing. If the "u" character is specified the listing will include a list of the modules taken from each the files specified on the command line. Any combination of these three characters may be specified.

-m=<symbol>
This option defines the primary function naming symbol. The primary function name is the external reference which the Linker attempts to resolve first. If left unspecified, the naming symbol defaults to "_main", which is usually the primary function in a C program. (At the C program level this primary function name is specified as "main", but the leading underscore is added by the Compiler, as is the case for all symbols generated by the Compiler. It is therefore important to remember that, when specifying a naming symbol that is contained in a compiled module, the symbol will always begin with a leading underscore.) The filename of the module which contains the primary function name is normally the name that will be assigned to any file(s) produced as a result of a Linker call line.

-n
This option prevents the Loader from being automatically executed when the Linker finishes. When the "-n" option is not specified, the Linker will normally default to calling the "standard" Loader (unless some other loader type has been optionally specified using the "-d(<c>]" option discussed previously).

-o=<file>
This option is used to assign a specific name, represented by <file>, to the Linker's output file. If this option is not used the output file will be given the same name as the module in which the primary function name is found. If no filename extension is explicitly specified, the Linker output filename will default to having a ".RL" extension.

-P[<C>]
This option is useful only an Unix-like operating systems, such as UNIX, INIX, and TNIX for example. On such systems, it causes the output of the Linker to be piped to the Loader rather than to be transferred in a temporary file. On some systems this

will cause a noticeable speed, improvement. The [<c>] indicates an optional character which may be used to specify that the Linker output should be sent to a particular optional Loader when use of the default "standard" Loader is not desired. The <c> character, when specified, represents the first letter in the Introl name of the desired Loader, just as for the case of the "-d[<c>]" option described previously.

-s

This option specifies that the output file is to be stripped of all non-entry defined symbols. This is useful when producing a partially linked module in which the user wishes to "hide" all the already resolved symbols. Partially linked modules are typically modules that have been linked, but not loaded, which may still contain unresolved references.

-t=<classlist>

This option is used to define an ordered listing of those non-zero class numbers, between 1 and 255, which are to be "preferred" for use in the linking process. The <classlist> can be a series of one or more numbers from "1" through "255". When a class list contains multiple class number entries, a comma or period must separate successive class numbers, as in "t=3,7,4", for example, which specifies the classes "3", "7", and "4". The order in which class numbers are entered on the link command line is significant to the Linker and defines the order of class preference. The first-entered (ie left-most) class appearing on the link command line will be given highest preference for inclusion by the Linker, the second-entered class will be given next highest preference, and so on. Modules of class 0 are always considered by the Linker as having lowest priority and are used in the linking process only if an identically named module having a class number which is included in the linker class list specification cannot be found by the Linker. For example, a class list such as "t=3,7,4" tells the Linker to preferably use modules of class 3 (if they can be found), or else use class 7 modules (if they can be found), or else use class 4 modules (if they can be found), or else, as a last resort, use modules of class 0 (if they can be found).

The reader is referred to the Loader Appendices of this manual for applicable Loader options that may be specified on the link command line.

LOADER

It is the Loader's function to fix absolute addresses for the relocated values in a relocatable module, thereby converting a relocatable module into an "executable" output file. The Loader is usually called automatically by the Linker but it may also be called separately by the user. As indicated below, several different Loaders are (optionally) available for use with Introl-C and, if the user has elected to obtain such optional Loaders, a variety of executable output file formats can be generated, depending on the Loader being used.

Each resident Introl-C compiler package, and each Introl-C cross-compiler package, nominally includes a single, specific type of Introl Loader which is considered as being the "standard" Loader for that compiler's particular host system configuration. For resident Introl-C Compiler packages, the 'standard' Loader that is furnished is an "operating system dependent" type of Loader which generates an output file that is executable on that particular Compiler's host system. For cross-compiler versions of Introl-C, the "standard" Loader furnished is typically a "hex" type Loader that generates a file of output load records, which can be either Motorola S-Records, intel Hex, Tektronix Hex, or Tektronix Extended Hex at user option. Besides the "standard" Loader that accompanies any given Compiler type, it is also possible for the user to optionally obtain and use other compatible "cross-Loaders" which generate output formats unrelated to the Compiler's host operating system. For example, "hex-type" Loaders are optionally available for use with resident versions of Introl-C; "operating system dependent" type Loaders are optionally available for use with cross-compiler versions; etc.

There are, therefore, several different species of Loaders, (as well as several different types of related Standard Libraries) that may potentially be used under Introl-C. The "standard" Loader supplied with your Introl-C package, as well as any other Loaders that may have been optionally ordered, are described in detail in the Loader Appendix of this Linker Reference Manual. This Loader section describes the general features that are common to all Loader types.

Normally the input to the Loader is expected to be a relocatable file which has no unresolved external references; if unresolved references do exist in its input, loading will normally not be successful. A Loader option is provided, however, to force a file to be loaded even if it contains unresolved references.

Usually a relocatable file has to be linked before it can be used as input to the Loader. It is also possible, of course, to assemble a file which makes no external references and then use the relocatable output file produced by the Assembler directly as input to the Loader (ie without having actually linked it).

LOADER COMMAND LINE

The "standard" Loader supplied with your Introl-C package (see Loader Appendices to this manual) is normally automatically called by the Linker when the Linker pass finishes. However, linker command line options exist (see Linker Section of this manual) that mav be used to alternatively force the Linker to automatically call other optional Loaders (assuming such optional Loaders have been obtained for use). Situations also arise when it is desirable to explicitly call the Loader alone, without first executing the Linker. When such situations arise, the Loader may be independently called by the user with a loader command line of the general form:

```
<c>ld <file> {<option>}
```

where <c>ld represents the Introl filename of the specific Loader being called, <file> is the name of the (linked) relocatable module which is to be loaded, and (<option>) represents zero or more Loader option specifiers.

Each of the potentially usable Introl Loaders is uniquely identified by a 3-letter Loader filename, the last two letters of which are always "ld". The <c> designator indicated in the "<c>ld" loader call on the command line therefore represents the first letter in the three-letter Loader name. For example, to call the Introl hex type of Loader, which has the filename "hld", the "<c>ld" entry on the command line would actually become "hld". For further specifics on the names of the loaders which can be legally accessed, refer to the Loader Appendices of this manual.

The relocatable file that is input to the Loader is expected to have a filenarne extension; if none is specified, the default filename extension ".RL" is assumed. Normally the name of the executable output file will be identical to the name of the input file, but with a filename extension typically added by the Loader. The filename extensions each Loader appends are discussed in the Loader Appendices to this manual.

LOADER OPTIONS

Each type of Loader available for use with the Introl-C has its own, generally unique set of options. The specific options that apply to each Loader furnished are discussed in the Loader Appendices.

When the Loader is being called separately, Loader options are specified directly on the loader command line when the Loader is being automatically called by the Linker, Loader options are specified on the link command line, together with the Linker options. If Loader options are specified on the link command line, any such options (ie those that do not apply to the Linker) will be automatically sent on to the Loader. For the most part Linker and Loader option specifiers tend to be distinct, so that there is little ambiguity when Loader options are specified on the link command line.

LIBRARY MANAGER

This section describes the features and operation of the Introl Library Manager.

For a program to be succesfully linked and loaded, all its external references must be resolved. That is, any functions which are referenced by the program but not included in the program must be added to it at link time. The Linker can be directed to search various files to find already compiled functions which satisfy these references. When it finds a piece of compiled code which satisfies a reference it includes the code in the resultant program. Any compiled or assembled file may be a legitimate input to the Linker. To facilitate the Linking process, it is often useful to have a file which contains more than a single piece of compiled code so that the user can specify a whole series of routines to the Linker with a minimum of fuss. Such a file is called a library file, an example of which is the introl-C Standard Library (libc.R). The Linker can search a library file and selectively extract only those modules it requires to link the file.

LIBRARY FILES

A library file is a file which contains one or more linkable object modules of the type produced by the Introl Assembler. When a file is compiled and assembled, the result is exactly one linkable module which is placed into a file. This file is actually a library which happens to contain only a single module. When the user links a program, one or more of these "libraries" are specified on the link command line. Usually the "libraries" are those produced as a result of a compilation and contain only a single module, however, they may also contain several modules. The Library Manager, "libman", is a program which allows the user to place several modules into a single library file. When the user has a large set of modules which are commonly used in programs, it is usually convenient to place them all in one library and then simply specify the library once on the link command line. The Linker will extract only those modules it requires in order to satisfy the external references of the program.

The Linker is designed to automatically search the "Standard Library", libc.R, if it still has external references to satisfy after it has exhausted all the alternatives provided by the modules specified an the link command line. For many C programs, the Standard Library is usually where most of the external references are satisfied. Many users find it useful to add to, or modify routines in, the Standard Library.

The Library Manager is the utility program which allows the user to create new libraries and also to maintain existing ones.

LIBRARY MANAGER

Because any file that is produced by the Assembler is already technically a library file, the Library Manager can correctly be looked upon as a program which manipulates libraries. Its input is a library file, such as a linkable object file produced by the Assembler. Thus, in the description below, references to "libraries" also implicitly includes those files output by the Assembler.

The Library Manager is called by entering a command line of the form:

```
libman <lib> {<optional-direct-command>}
```

where <lib> is the name of the library to be edited and <optional-direct-command> is an optional command to the Library Manager. If the <optional-direct-command> entry is omitted, the Library Manager will enter its "Interactive Mode" of operation and solicit library management commands from the user terminal.

The input library specified by <lib> may be either a new library or an existing one and, unless the user takes contrary action, it will also be the nane of the output library.

MODES OF OPERATION

The Library Manager has three modes of operation: Direct Mode, Interactive Mode, and Command File Mode. The most convenient to use for simple additions and deletions to the library is the Direct Mode. For more extensive modifications the user may instead wish to use Interactive mode. The third mode is the Command File mode which causes the Library Manager to read its commands from a file rather than getting them from the user terminal.

Direct Mode: In Direct Mode the user is permitted to specify a single command on the library manager command line. When the Library Manager is called, it executes this single command function and then immediately exits from the Library Manager. When modifying libraries, however, a single command function is often all that is necessary to accomplish the change desired by the user. When Direct Mode is being used, the desired command is specified right on the command line, following the <lib> library specification. Any Library Manager command may be used in the Direct Mode.

Interactive Mode: if no command is specified on the Library Manager call line, the Library Manager will enter its Interactive Mode of operation. In Interactive Mode the Library Manager will print a colon (".") as a promet and will accept a succession of commands directly from the user terminal. Interactive Mode is useful when the user must make extensive changes to a library, or when the user wishes to step through the library checking and/or changing modules in an "interactive" manner. Once selected, the Interactive Mode will remain in effect until the user enters a "quit" or "omit" command.

Command File Mode: One of the commands which the user can specify as an Interactive code or a Direct Mode command entry is the "Comfile" command. This command instructs the Library Manager to read subsequent instructions from a command file. When a "Comfile" command is entered, the Library Manager will read from the file specified until it reads a "quit" or "omit" command or, alternatively, until it reaches the end of the file. when exiting the Command File Mode, the Library Manager will return to whatever mode it was in before the Command File Mode was entered. If the Command File Mode was entered as the result of a Direct Mode command, then the Library Manager will terminate when Command File Mode is exited. If entered from the Interactive Mode, it will return to the Interactive Mode.

LIBRARY MANAGER COMMANDS

In the descriptions that follow, the commands may be abbreviated to the characters shown in capital letters. For simplicity, the descriptions are specified in a BNF type form. In this form items enclosed in angle brackets "<" and ">" represent names or numbers to be chosen by the user. Items enclosed in square brackets "[" and "]" represent optional items. Anything enclosed in curly brackets "{" and "}" may be repeated zero or more times. These "meta" characters (ie <,>,{,},[, and]) are just to help the user understand what is required and should not actually be typed in. Thus the "delete" specification ...

```
Delete {<module>{,<class>])
```

means that the delete command (which may be abbreviated to just "d") requires zero or more user-specified module names, each of which may have an optional class specifer which is separated from the module name by a comma.

In the following:

<module>	refers to the name of a module (which should consist of a series of characters). The first character may not be a digit.
<file>	refers to any legal file or path name.
<class>	is a number from 0 to 255 which represents a module's class number.

Thus a legal example of the delete command could be:

```
d modulea,2 moduleb modulec, 0
```

which would cause three modules to be deleted; the class 2 "modulea" module, the class 0 "moduleb" module, and the class 0 "modulec" module.

Add {<file>{,<module>f,<class>]}}

The add command is used to add modules to an existing library or to create a new library. It consists of the word "add", which may be abbreviated to "a", followed by one or more filenames, each of which may be followed by zero or more module specifications, each of which may include a class specification. It is possible to add modules at a specific place in the library (see the "find", command) but for most linking applications it makes no difference where a module is located in the library. In Direct Mode, the add command will add modules to the end of a library. In Interactive Mode or Command File Mode, the Library Manager can be directed to add a module anywhere in a library. The argument to the add command is a filename which should contain at least one linkable module (such as that produced by a compilation). The filename may be followed by any number of module names. If there are no specifications following the file name, the Library Manager will attemct to add all of the modules contained in the file. If specific nodules are named, the Library Manager will attempt to add only those modules from the named file. Any module may have an optional class specification, which is a numeric specifier in the range of 0 to 255. If the class specification is not present, the first module encountered having the specified module name, regardless of its class, will be added to the library; otherwise only a module with a matching name and class will be added. The add command will not add any module whose name and class match one already existing in the library.

Delete {<module>{,<class>]}
This command allows the user to delete modules from a library. The delete command will attempt to delete the named modules, taking into account the module's class, if it is specified. If the class specifier is omitted, and there is more than one module having the specified name in the library, the delete command will print a warning message and will not delete the module. The user may then delete the module by specifying the class of the module which is to be deleted.

The delete command will print a warning message if no module name is specified.

Revlace {<file>{,<module>{,<class>]}}
The replace command is used to replace modules in an existing library. It consists of the word "replace", which may be abbreviated to "r" followed by one or more filenames, each of which may be followed by zero or more module specifications, each of which may include a class specification. The argument to the replace command is a file name which should contain at least one linkable module (such as that produced by a compilation). The filename may be followed by any number of module names. If there are no module specifications following the file name, the Library Manager will attempt to replace all of the modules contained in the file. If specific modules are named, the Library Manager will attempt to replace onlv those modules. Any module may have an optional class specification. If the class specification is not present, the first module with a matching name, regardless of its class, will be replaced in the library; otherwise only a module with a matching name and class will be replaced. The replace command will only replace a module whose name, or name and class (if both are specified), match a module already in the library.

Quit
This command quits the Library Manager, first saving the library file if it has changed. This command may be abbreviated to "q".

OMIT
This command directly exits the Library Manager without saving the library that was being edited. You may want to remember this one in case you hopelessly mess up a library file (although that shouldn't be cause for panic since the Library Manager always makes a backup file). Notice that there is no abbreviation for this command.

List {<module>{,<class>}}
The list command will print out information an the named modules. If no modules are specified, the list command will print out information on all of the modules in the library.

SList {<module>[,<class>]}
This is a short form of the List command. It prints out an abbreviated listing containing only the module name, class, and revision of each named module. If no modules are specified, this information will be printed for all modules in the library.

Help
The help command allows the user to obtain on-line help when using the Library Manager. It assumes there is a help text file available. The help command will print a menu and request a number from the user; it then prints the associated message and enters Interactive Mode.

LOad {<file>}
When anything is done involving a library which is currently not in memory, it is automatically loaded. The "load" command may be used to explicitly load a library without actually doing anything with it. Loaded libraries are not the same as the library you are editing; it is simply a library whose module information is in memory. When a module is from a library, for example, the module information for the entire library is loaded into memory so that the Library Manager can more quickly reference it. Before a file is loaded, the memory is checked to see if the file has already been loaded. A file is never loaded more than once. The "load" command may be abbreviated to "lo".

The reason a user may want to load a library explicitly is so the contents of a loaded library may be listed and examined using the load-list command as described below.

LList {<file>}
The LList command allows the user to list a loaded library. When used with a library name, the LList command will list the contents of the named library. When specified without any library name the LList command will list the names of all the currently loaded libraries. The "llist" command may be abbreviated to "ll".

SLList {<file>}
This command provides an abbreviated load-listing, including only the module name, class, and version. When this command is used without any library name specified, it will list the names of all currently loaded libraries. The "sllist" command may be abreviated to "sll".

Save {<file>)
The save command will force the Library Manager to save the library using the filename indicated by <file>. If no filename is explicitly specified, the library will be saved using the library name originally specified on the command line. As a safety measure, any time a file is saved the Library Manager will make a backup copy of any file which would have been overwritten by the save process. It will append a ".bak" extension to this backup file. The Library Manager will automatically save the library whenever the user exits using a "quit" command.

Comfile {<file>}
This command will direct the Library Manager to execute commands read from one or more specified files until it reads a "quit" or "omit" from the specified files or, alternatively until the end of the file is reached. An error message will be printed if no file is specified. The "comfile" command may be abbreviated by "c".

Echo {<any-string>}
This command simply echos the specifed strings to the terminal. This command can be useful in a command file to inform the user of its progress.

INTeractive
This command will explicitly place the Library Manager in Interactive Mode. Needless to say, it has no use when already in the Interactive mode, and very little use as a Direct Mode command (since the user can more readily enter Interactive Mode by simply not specifing any command whatever when calling the Library Manager). It is potentially useful in the Command File Mode, however, and can be included in a command file to force a return to the interactive Mode. The "interactive" command may be abbreviated as "int".

Find {<module>{,<class>]}
This command is used to "find" the module whose name and class is given.

There is a pointer in the Library Manager which points to what is known as the "current" module. When the Library Manager starts, the "current" module is the last-occurring module in the library being edited (assuming there are any modules in the library being edited). When an "Add" command is executed for example, the newly added modules are added following the "current" module. Almost every command has some effect on which particular module in the library is considered as being the "current" module after the commanded action has been completed. Following an add command, for instance, the "current" module will become the last module that was added because of that add command. The list command also causes the current module to become the last module that is actually listed. In this manner, user command inputs continuously alter which specific module is actually considered the "current" module at any give time.

The find command can be used to explicitly define the current module to be any specific module in a library. Thus, if the user wishes to place a module in a specific place within the library, he can "find" the module which is to immediately precede the new module, and then "add" the new module. This will cause the new module to be placed immediately after the module that was "found" using the find command; this, of course, would also cause the "current" module to then become the newly added one.

The find command will attempt to move the "current" module pointer to the named module. It starts searching from the current module and continues until it reaches the bottom of the file, at which point it starts searching from the top of the file. It searches in this manner until it finds the named module, or until it reaches the original current module. If no module class is specified, the find command will stop at the first module it encounters that has the specified module name, regardless of its module class number; otherwise it will attempt to find a module which has both the name and class specified in the find command.

Print {<module>[,<class>]}
This command causes information to be printed for the named modules. If no modules are specified, it will print information on the "current" module.

SPrint {<module>[,<class>]}
This command works just like the Print command except it prints an abbreviated listing which includes only the name of the module, its class, and its revision.

Insert {<file>{,<module>f,[class>]}}
This command is similar to the "Add" command except, rather than placing the named modules after the "current" module, it will place them proceeding the current module in the library. When the Insert function finishes, the last module that was inserted then becomes the current module.

Stepping Through The Library

When editing a library using the Library Manager, a pointer exists which indicates the "current" module (as was described previously under the "find" command). This pointer is used as a starting point for searches when adding, exchanging, and deleting modules. It also points to the module which will be printed out by a "print" command when print is used without arguments. Most of the commands affect the value of this pointer, usually leaving it pointing to the last module that was referenced. There are several ways for the user to change the "current" module pointer. One is via the "find" command (see the Find command, above). For example, the following command moves the painter to a module named "thing":

```
find thing
```

The user may also move the current module pointer around in a "relative" fashion by specifing a signed integer on the line. For example, the following will move the pointer backwards four (4) modules:

```
-4
```

By comparison an entry such as:

```
+2
```

will move the pointer forward two (2) modules.

It is also legal to specify one or more successive minus ("-")or plus ("+") signs to indicate the total number of modules to move backward or forward. For examole, a single minus or plus sign would move the pointer backward or forward one module, respectively. Two minus or two plus signs will move the pointer backward or forward two modules respectively (one for each symbol), and so on. It is also legal to move the pointer to a module located an absolute number of modules from the begining of the library; this is done by entering an unsigned number. For example, entering:

```
12
```

will move the pointer to the twelvth module in the library.

Any time one of these commands is executed, the Library Manager will print the name of the resultant current module. If one of these commands attempts to move the "current module pointer" above the top or below the bottom of the library, the Library Manager will print "TOP" or "BOTTOM" respectively.

CRstep
Executing this command toggles a flag which, when "on", causes a carriage return to act like a plus ("+") sign. This then allows a user to step down through the library, one module at a time, by simply hitting the carriage return. The CRstep command toggles this feature on (if previously off) or off (if previously on) with each execution. Therefore, if this feature has been previously selected to be "on", it can be selected to be "off" by simply re-entering the CRstep command once again.

QUIET
This command will prevent the Library Manager from printing out the name of the current module when the "current module" pointer moving commands are used. The "quiet" command may be abbreviated by "quie".

Additional Notes
If the user wishes to write out a module which is in a library, this

can be easily done by a command of the type:

```
libman newmod add oldlib,mod
```

For the filenames used in this example, this instructs the Library Manager to make a new library, called "newmcd", which contains a single module, called "mod", which was obtained from a library called "oldlib".

APPENDICES

This section contains miscellaneous reference information which may be useful to the programmer.

APPENDIX A

LINKABLE FILE FORMAT

The following is the linkable file format which is expected by the Introl Linker and Loader.

There is no difference between a library file and a linkable object file as produced by the Assembler, other than the fact that a linkable object file contains only a single module whereas a library usually contains multiple modules. In the special case of a file which contains only a single module, it is permissible to have a text size specified as zero even though the text has a non-zero length. When a multi-byte value is specified, the most significant byte is assumed to appear first.

INTROL LINKABLE BINARY FILE FORMAT

```
HEADER
        2 bytes Magic #
        2 bytes Number of module descriptors in this file
        1 byte Checksum of header

MODULE DESCRIPTOR (repeated for each module)
        4 bytes Offset to module text in file
        4 bytes Size of text (may be zero if
          single module in this file)
        2 bytes Size of string area
        1 byte   Module class
        1 byte   Module revision
        4 bytes  Relocatable segment @ax sizes

                 |SF|SE|...|S7|S6|...|S0|

                 Sn is a two bit max size  specifier:
                         00   one byte max size
                         01 - two byte max size
                         10 - three byte max size
                         11 - four byte max size

          4 bytes Relocatable segment  size descriptors

                  |SF|SE|...|S7|S6|...|S0|

                  Sn is a two bit descriptor size value:
                          00 - no size
                          01 - one byte size
                          10 - two byte size
                          11 - four byte size

           { 0..4 bytes segment 0 size }
           { 0..4 bytes segment 1 size }
                         .
                         .
                         .
```

```
{ 0..4 bytes segment F size }

2 byte symbol count

For each symbol up to symbol count:

   2 bytes Offset of identifier in string area
   2 byte Descriptor value

     |SZ|XXXXX|N|E|I|R|A|SEGM|

     SZ      is  the  descriptor  of  the  symbol's value
             00 - the value is zero
             01   -  the   value  follows   in  one  byte
             10   -  the  value   follows  in  two  bytes
             11   -  the  value  follows  in  four  bytes

     X    is reserved
     N    set if the symbol is an entry point
     A    set if the symbol is absolute
     E    set if the symbol is exported
     I    set is the symbol is imported
          (both E and I are set if the symbol
          is undefined segment imported)
     SEGM  is  the  segment  the  symbol  resides  in  if
          non-absolute.

  { 0..4 byte symbol value }

The module descriptor string area starts here.  The strings in
the string area are null terminated ASCII character strings.
The first string in the string area is the module name.
```

PROGRAM TEXT (follows all module descriptors in the file)

```
The basic text format is:

        |CM|MODIFY| { 0 or more operand bytes }

                CM     is the two bit command.
                MODIFY is 6 bits of command specific info.


code 00 - Special function

        |00|FNCODE| {|function specific operands|}

        FNCODE   is  a   six  bit   special  function  code:

           0 - end of text
           1 - set byte size relocation
           2 - set word size relocation
```

```
          3 - set long size relocation

          codes 4-15 are Loader commands

          4   -reserved
          5   -reserved
          6   -    "
          7   -    "
          8   -    "
          9   -    "
          10  -    "
          11  -    "
          12  -    "
          13  -    "
          14  -    "
          15  -    "
  Multiple byte commands
     The byte count is represented in the lower
     two bits as  follows:
              00 - the byte count follows in one
                   byte
              01   -  the  operand   follows  in  one  byte
              10   -  the  operand  follows  in  two  bytes
              11  -  the  operand  follows  in  four  bytes
          16  - reserved
          17  - skip with one byte byte count
          18  - skip with two byte byte count
          19  - skip with four byte byte count
          20  - reserved
          24  - reserved
          28  - reserved

                  Segment set commands

          32  - set segment 0
          33  - set segment 1
          34  - set segment 2
           .
           .
          46  - set segment E
          47  - set segment F
          48  - reserved
          49  -     "
           .
           .
          63   reserved


   coce 01 - pass absolute text
|01|TCOUNT| |TCOUNT bytes of text|
        TCOUNT - is the number of bytes to pass
                 (1-64).  If TCOUNT == 0 then
                  byte count is 64.

 code 10 - offset relocation command

|10|R|X|SEGM|  |relocation size offset|
```

Relocation is done in the previously specified relocation size. The result is the proper relocated datum with the base of the given segment in this module added to the following offset. If the relative bit is set, the result is the proper relocated datum with the result being equal to the relocated value minus the value of the location counter follow- the relocated value.

```
        R    - set if the relocation is relative
        X    - is reserved
        SEGM - is the segment # to relocate with

code 11 - symbol relocation command

|11|R|XX|S|OF |one or two byte symbol #|  {|offset|}
```

Relocation is done in the previously specified relocation size. The result is the proper relocated datum with the result being equal to the value of the re- solved symbol plus the optional following offset. If the relative bit is set, the result is the proper relocated datum with the result being equal to the relocated value minus the value of the location counter following the relocated value.

```
        R - set if the relocation is relative
        XX - reserved
        S  -  0 if  one  byte symbol  #, 1  if  two byte  sym. #
        OF - size of the following offset

              00 - zero offset
              01 - byte offset
              10 - word offset
              11 - long offset
```

APPENDIX LF

FLD LOADER
OPTIONS AND RUNTIME ENVIRONMENT

The Introl Loader which generates Flex format output files is called the "fld" Loader.

The fld Loader is the "standard" Loader that is furnished with the part number FC6809 Introl-C Compiler and, as such, is the the Loader normally called by the FC6809's Linker when it finishes linking. The fld Loader is also optionally available for use with other versions of Introl-C (ie for Introl-C packages that do not themselves run under the Flex operating system) and, in such cases, is considered as being an "optional" Loader for these versions. (Refer to the "-d[<c>]" option discussed in the Linker section of this manual.)

The loader command line call for the fld Loader is of the form:

```
fld <filename> {<options>}
```

where <filename> is the module to be loaded and <options> are zero or more fld Loader option specifiers.

The fld Loader expects its input to be a relocatable module as produced by the Introl Linker, with any applicable "standard library", references having been being resolved using the FC6809 Standard Library. The fld Loader produces an output that is compatible with, and executable under, the Flex operating system. Executable files generated by the fld Loader are characterized by the filename extension ".CMD", which the fld Loader automatically appends to its output file.

Unless otherwise indicated, the following options for the fld Loader may be specified on either the linker command line (the typical case when the Loader is being automatically called by the Linker) or on the loader command line (when the Loader is being called independently by the user).

OPTIONS

-a=<sec>:<seg>{,<seg>}
: Assign segment to a section; where <sec> represents a Flex program segment which should be either "text", "data", or "bss", and <seg> is a segment number in the range 0 to 15. This option allows the user to override the default settings for placement of program segments.

-c=<file>
: Get additional parameters from a command file; where <file> is the command file filename. This option allows the user to specify an unlimited number of parameters by placing them, one to a line, in the named text file.

-l[s][=<file>]

Produce an output listing; where the "s" character is an optional entry, and <file> is an optional filename. This option forces the Loader to generate an output listing. If the optional s character is specified, the listing will contain symbol information. If the optional filename specification is included, the listing will be placed in the named file.

-o=<name>

Set output file name; where <name> is to be the name of the output file. If this option is omitted, the output file name will be that of the input name. If no filename extension is explicitly defined, the default extension ".CMD" will be assigned.

-W

Make an executable file no matter what! This option will cause the Loader to produce an executable output file even if there are still unresolved external references. It is not guaranteed as to what the result will be if the program actually attempts to access one of these unresolved items.

-y[{t|d|b}]=<origin>

Set origin; where the "t" or "d" or "b" character is optional, and <origin> is a hexadecimal number. This option may be used to set the origins of the text, initialized data, and uninitialized sections of the output file. If no t or d or b character is specified, or if the t character is specified, the text section will be placed at the location indicated by <origin>. If the d character is specified, the initialized data section will be placed at the location indicated by <origin>. If the b character is specified, the bss (uninitialized data) section will be placed at the location indicated by <origin>. if this option is not specifed, the text section will default be being placed at the zero origin, and will be immediately followed by the initialized data section, which will be immediately followed by the uninitialized data section.

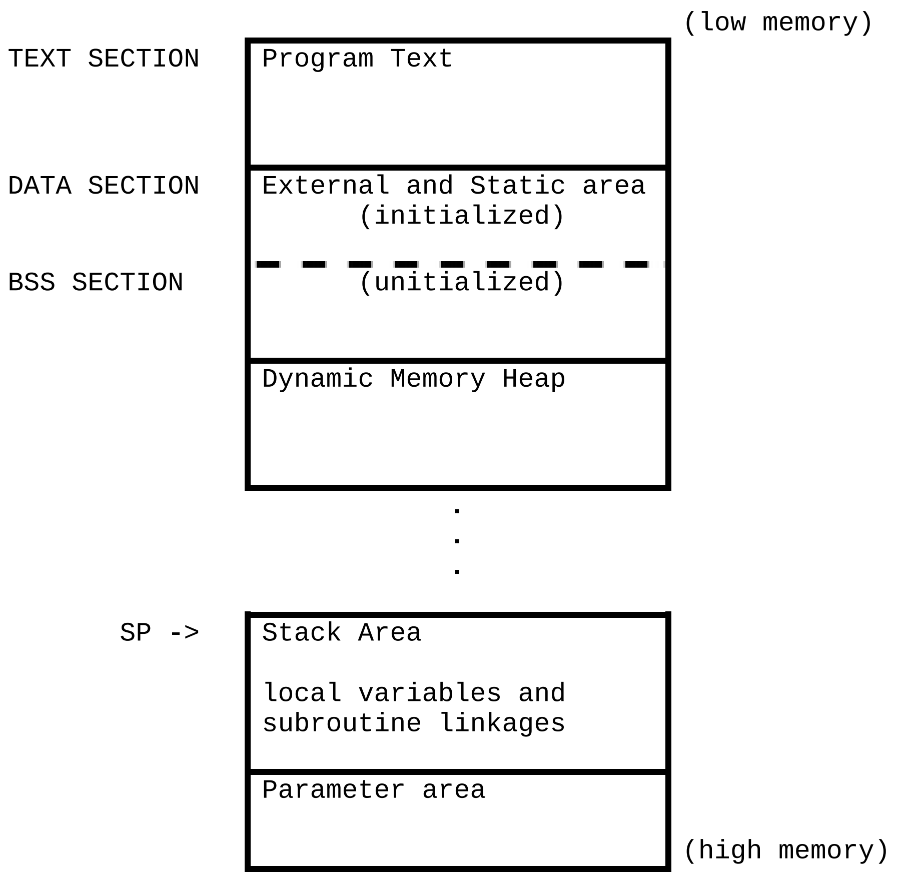
-Z

Zap the input file. This option deletes the input file after the Loader has finished using it. When the Linker automatically calls the Loader, the Linker normally specifies this -z option as part of the call to cause the Loader to delete the file produced by the Linker (ie the intermediate ".RL" extension file) when it is no longer needed for loading purposes.

RUNTIME DATA MEMORY MAP

The runtime memory map shows the layout of the data space which a program has available during execution. The data appears in two areas, one of which is placed toward the low end of memory and another which is placed at the high end of memory (below the Flex operating system). The heap is placed in the low end of memory and grows upward by asking the operating system to enlarge its memory space. The stack is placed in the area at the high end of memory.

DATA MEMORY MAP



Introl-C is a registered trademark of Introl Corp.
Flex is a trademark of Technical Systems Consultants, Inc.

INDEX

INTROL-C/6809

C LANGUAGE DEVELOPMENT SYSTEM

Introl-C/6809 is a powerful C language compiler system that is designed to facilitate the development of high-efficiency software for the 6809. The Introl-C package includes a C Compiler, 6809 Relocating Assembler, Linker, Loader, Library Manager, and Standard Library. It has been in the field since early 1982 and has gained widespread acceptance among users for its reliable and comprehensive support of the C language as well as its ease of use. Its ability to generate exceptionally compact, fast executing code has long distinguished the Introl-C implementation as being the most efficient high level language that is available for the 6809 and has resulted in Introl-C's widespread use in the industrial community for development of process control software. Programs developed using Introl-C are typically within 15 to 20% or less of the size and speed of programs written entirely in 6809 assembler. For the particular case of the Eratosthenes Sieve Benchmark, the p/n UC6809 resident Introl-C Compiler, for example, produces a 176 byte compiled module, a total program size of 2007 bytes, and a program execute time of 8 seconds on a 2 Mhz 6809.

Code produced under lntrol-C is re-entrant, relocatable, and ROMable and may be installed on any 6809 target, including ROM-based systems. No fees or royalties of any type are imposed on object code programs developed using the compiler. Introl-C/6809 is available as resident software for 6809-based hosts running UniFlex, Flex, or 0S9. Cross-software versions of Introl-C are available for PDP-11 based hosts running UNIX (or any of the URIX look-alikes such as TNIX, VENIX, etc), PDP-11 based hosts running RSX11M, and also for IBM PC hosts running PC DOS or MDOS.

Introl-C is designed to the standard C language specification defined by Kernighan & Ritchie and supports all features of the language except fields, doubles, and the #if and the #line preprocessor directives (all other preprocessor directives, including #ifdef and #ifndef, are fully supported). Extensions to the standard language include provisions to permit nesting of comments, use of separate name spaces for all union and structure member names, and provision to allow symbol names up to 90 significant characters in length. Most C source programs developed using Introl-C are directly usable as input to standard UNIX C compilers.

User interface is designed for case of use permitting C source files to be converted into executable outputs with a minimum of effort on the part of the programmer. For example, a single command line entry of the form:

```
icc filename (options]
```

will cause a C source file to be fully compiled and assembled to produce a relocatable object module. Similarly, relocatable modules may be linked and automatically loaded to produce an executable output via a simple command line entry of the form:

```
ilink filel [file2 file3 ...] [options]
```

Numerous options are supported to permit versatile, user-controlled alteration of the standard compilation, assembly, linking and loading processes. Option specifications, however, are generally required only for specialized circumstances since the defaults for unused options are designed to select standard conditions that "make sense" for the great majority of program development situations.

The C Compiler is a 4-pass program that generates an optimized assembly language file as its output. In normal use, the 4 sequential compilation passes execute automatically and are followed by automatic execution of the included Assembler, thus resulting in a fully compiled, fully assembled relocatable object module as the result of a typical compiler call. The intermediate assembly language file generated by the C Compiler is available to the user, however. Compile time options include capability for the user to selectively place data of a given type under any of 16 different location counters (ie "segments"), ability to generate either position-dependent or position-independent code and/or data, and capabilities for specifying #define pre-processor directives directly on the compiler call line. The 16 location counters provided by Introl-C, and the features for generating either position-dependent or position-independent code and data, allow significant flexibility when finished programs are to be placed in ROM.

The included R09 Assembler is a full-featured 6809 Relocating Assembler. Although nominally furnished to provide automatic assembly of the C Compiler's output, the Assembler may also be called directly by the user for converting user-written assembler text files into relocatable object modules. The Assembler supports all addressing modes of the 6809, recognizes all standard opcodes, and will accept arbitrarily complex assembly-type input expressions. A unique feature of the Introl Assembler is its assignment of a user-definable "class" identifier to each module it produces which is used (in combination with the module's name) for identification purposes by the Linker. This feature allows the user to create (and the Linker to distinguish between) multiple versions of identically-named C support functions within the Standard Library, for example, and provides the basis for a powerful and convenient link-time capability for tailoring the link process to minimize runtime overhead in the final program.

The Linker accepts any number of relocatable modules as input and produces a single relocatable module as its output. Multiple-pass linking is supported. The Linker also supports "partial linking", wherein several component modules of an overall program may be linked together to form a single resultant "partially linked" module which may then be reused as input in subsequent linking operations. Linked programs up to 64K in size may be produced.

Although a specific host-related Loader is supplied with each Introl-C software package, any of four different types of Introl Loaders are potentially usable with any of the Introl-C compiler systems: a FLD Loader, which generates a file of output load records in hex format (Motorola S Records, Intel Hex, Tektronix Hex, or Extended Tektronix Hex formats); a ULD Loader, which generates outputs that are executable under UniFlex; a FLD Loader, which generates outputs executable under Flex; and an OLD Loader, which generates outputs executable under 0S9. The loader supplied with P/N XC6809, RC6809, and PC6809 cross-software packages is the HLD Loader; the ULD Loader is supplied with the P/N UC6809 package; the FLD Loader is supplied with the P/N FC6809 package; and the OLD Loader is supplied with the P/N 0C6809 package. Any of these several Loader types, however, is also optionally available, at extra cost, as a "cross-loader" for use with any resident or cross-software version of Introl-C. The HLD loader is particularly useful when developing software for standalone, ROM-based applications.

The Standard Library contains an extensive collection of C programming support functions, including I/0 and arithmetic functions not directly performed by the 6809. Only those library functions actually required for program execution are extracted by the Linker, resulting in minimum runtime overhead in developed programs. The library furnished with each of the several resident versions of Introl-C is host-O.S.-specific in nature whereas the library supplied with cross-software versions is operating-system-independent and is tailored for use in standalone-target applications. Standard Library source code is available as an extra cost option.

The Library Manager provides convenient and versatile capabilities for adding. deleting. and modifying Standard Library functions, thus permitting unique libraries of C support functions of any type to be created by the user. All library functions are stored in linkable format, thus avoiding any need for recompiling them before each use. The Library Manager, in combination with the Linker, significantly reduces development times for large programs by allowing individual parts of the program to be independently developed and compiled with a minimum of effort.

3/1/84

M09

6809 RESIDENT AND CROSS MACRO ASSEMBLERS

Introl's M09 resident and cross macro assemblers are designed to translate MC6809 assembly language source programs into 6809 machine code. All M09 assembler packages include the Introl M09 Relocating Macro Assembler, the ILINK Linker, a Loader, and the LIBMAN Library Manager. Resident versions of the M09 software package are available for use on 6809-based microcomputers running under UniFlex, Flex, or 0S9. Cross-software versions are available for use on PDP-11 based development systems running UNIX or RSX-11M, and also for IBM PC hosts running PCDOS.

The Assembler is a full-featured relocating macro assembler that accepts a 6809 assembly language text file as input and produces a relocatable object file as its output. The included Linker and Loader permit any number of assembled modules and/or library modules to be linked together and then loaded to produce a single resultant output file in an executable format. The Library Manager provides convenient and versatile features for the user to create, and maintain, libraries of assembled modules. Resident assembler packages incorporate a host-specific Loader (ULD, FLD, or OLD, as applicable) that produces output files which are executable under the assembler's host operating system. M09 cross-assembler packages incorporate a hex-type Loader (HLD) which generates a file of output load records in any of several hex formats: Motorola S Records, Intel Hex, Tektronix Hex, or Extended Tektronix Hex format, at user option. Introl's HLD, ULD, FLD, and OLD Cross-Loader packages are optionally usable with any version of M09, and are available from Introl at extra cost. The M09 Macro Assembler is fully compatible, both in source format and object output, with the R09 Relocating Assembler that is supplied with the Introl-C/6809 C Compiler system.

M09 supports macros and conditional assembly, recognizes the standard opcodes recognized by Motorola assemblers, and supports the complete instruction set and all addressing modes of the 6809. The Assembler accepts assembly type input expressions that are arbitrarily complex. Parentheses are allowed in expressions to modify the evaluation order of operators. Symbols may be of any length, with the first 100 characters being significant and retained by the assembler. Assembly time expressions may be used in the operand of any assembler opcode or directive. Symbols and constants may be used interchangeably in an expression. All results of expressions at assembly time are 32-bit truncated integers.

The following assembly time options are supported by the M09 Assembler:

-a	Place all symbols except those beginning with a "?" character in the object file.
-c	Send Assembler's output listing to console.
-f	Force listing of all conditionally excluded code.
-i	Include all included files in output listing.
-j	Include symbols beginning with a "?" character in the output listing.
-l=(filename)	Place output listing in specified file.
-m	Include all macro-expansion-generated code in the output listing.
-n	Don't produce an output listing.
-0=(filename)	Explicitly assign name to the relocatable output file.
-p(digit)=(value)	Pass parameters to macro program being assembled.
-q=(class)	Assign numeric class identifier to the relocatable output module.
-s	Suppress listing of the symbol table.
-U	Force all undefined symbols to default to imported symbols.
-X	Don't generate an object module.
-y=(pathname)	Search "pathname" for macro files after searching current working area.
-z	Delete input file after Assembler has finished using it.

Assembly time expression operators supported:

-	unary minus (twos complement)	&	bitwise and
~	not (ones complement)	^	bitwise exclusive or
*	multiplication	|	bitwise inclusive or
/	division	>	greater than
%	mod (remainder)	<	less than
+	addition	>=	greater than or equal to
-	subtraction	<=	less than or equal to
<<	shift left	==	equal to
>>	shift right	!=	not equal to

Assembler directives supported:

comm	Common Area	if	Numeric Conditional Assembly
dc	Define Data Constant	ifn	Numeric Conditional Assembly
ds	Define Data Storage	ifc	String Compare Conditional Assembly
else	Conditional Assembly Else	ifnc	String Non-Compare Conditional Assembly
endif	End Of Conditional Assembly	import	External Symbol Reference
endm	End Of Macro Text	lib	Load A Disk File
end	End Of Assembly	list	Terminate Previous Nolist Directive
equ	Equate Symbol With A Value	loc	Select Location Counter
err	Prograrmer-Generated Error	macro	Define A Macro
exitm	Exit From Macro	nolist	Turn Off The Listing
export	External Symbol Definition	offset	Absolute Offset From An Origin
fcb	Form Constant Byte	repeat	Repeat The Next line
fcc	Form Constant Character	rmb	Reserve Memory Bytes
fdb	Form Double Byte Constant	set	Set Symbol To A Value
ident	Identify Module	syn	Equate Labels

....

3/1/84

INTROL-C PRICE LIST

PART NO.	PRODUCT DESCRIPTION	PRICE	1-YEAR MAINTENANCE DOMESTIC/FOREIGN

INTROL-C/6809 COMPILER PACKAGES

(All packages include ICC Compiler, R09 Assembler, ILINK Linker, LIBMAN Library Manager; resident compilers include applicable host-compatible Loader and Standard Library; cross-compilers include HLD Loader and STA09 Standard Library.

PART NO.	PRODUCT DESCRIPTION	PRICE	1-YEAR MAINTENANCE DOMESTIC/FOREIGN
UC6809-UFX09	Resident compiler package for 6809/UniFlex host	$425	$100/$135
FC6809-FIX09	Resident compiler package for 6809/Flex host	$425	$100/$135
0C6809-05909	Resident compiler package for 6809/059 host	$425	$100/$135
XC6809-UNXI1	Cross-compiler package for POP-11/UNIX host	$2500	$500/$550
RC6809-RSXII	Cross-compiler package for POP-11/RSX1IM host	$2500	$500/$550
PC6809-PCDOS	Cross-compiler package for IBM PC/PCDOS host	$750	$200/$235
MAN-C6809	Manual only (Specify compiler type)	$75	NA
MANEX-C6809	Additional Manual (Specify compiler type)	$35	NA

....

LIBRARY SOURCE CODE PACKAGES

PART NO.	PRODUCT DESCRIPTION	PRICE	1-YEAR MAINTENANCE DOMESTIC/FOREIGN
UFXO9LS-(*)	Source code for UC6809 Standard library	$400	$100/$135
FLX09LS-(*)	Source code for FC6809 Standard Library	$400	$100/$135
0S909LS-(*)	Source code for 0C6809 Standard library	$400	$100/$135
STAO9LS-(*)	Source code for XC6809/PC6809/RC6809 (Standalone) Library	$400	$100/$135

(*Note: Specify host O.S. format desired; ie whether UniFlex, Flex, 0S9, UNIX, RSX11M, or PC DOS)
....

CROSS LOADER PACKAGES

(HLD Loaders include STA09 Standalone Library; others include targeted-host-compatible Library)

PART NO.	PRODUCT DESCRIPTION	PRICE	1-YEAR MAINTENANCE DOMESTIC/FOREIGN
HLD-UFXO9	HLD (hex format output) Loader for UC6809 compiler	$150	$30/$40
HLD-FLX09	HLD (hex format output) Loader for FC6809 compiler	$150	$30/$40
HL0-OS909	HLD (hex format output) Loader for 0C6809 compiler	$150	$30/$40
ULD-FLX09	ULD (UniFlex format output) Cross-loader for FC6809 compiler	$150	$30/$40
ULD-OS909	ULD (UniFlex format output) Cross-Loader for 0C6809 compiler	$150	$30/$40
ULD-UNXI1	ULD (UniFlex format output) Cross-Loader for XC6809 compiler	$300	$75/$90
ULD-PCDOS	ULD (UniFlex format output) Cross-Loader for PC6809 compiler	$225	$50/$65
ULD-RSX11	ULD (UniFlex format output) Cross-Loader for RC6809 compiler	$300	$75/$90
FLD-UFXO9	FLD (Flex format output) Cross-Loader for UC6809 compiler	$150	$30/$40
FLD-OS909	FLD (Flex format output) Cross-Loader for 0C6809 compiler	$150	$30/$40
FLD-UNXI1	FLD (Flex format output) Cross-Loader for XC6809 compiler	$300	$75/$90
FLD-PCDOS	FLO (Flex format output) Cross-Loader for PC6809 compiler	$225	$50/$65
FLD-RSX11	FLO (Flex format output) Cross-Loader for RC6809 compiler	$300	$75/$90
OLD-UFXO9	OLD (OS9 format output) Cross-Loader for UC6809 compiler	$150	$30/$40
OLD-FLX09	OLD (OS9 format output) Cross-Loader for FC6809 compiler	$150	$30/$40
OLD-UNX11	OLD (OS9 format output) Cross-Loader for XC6809 compiler	$300	$75/$90
OLD-PCDOS	OLD (OS9-format output) Cross-Loader for PC6809 compiler	$225	$50/$65
OLD-RSX11	OLD (OS9 format output) Cross-Loader for RC6809 compiler	$300	$75/$90

PART NO	PRODUCT DESCRIPTION	PRICE	1-YEAR MAINTENANCE DOMESTIC/FOREIGN
6809 MACRO RELOCATING ASSEMBLER PACKAGES			
(All packages include M09 Macro Relocating Assembler, ILINK Linker and LIBMAN Library Manager; resident assemblers include host-compatible Loader; cross-assemblers include HLD Loader.)			
M09-UFXO9	6809 Macro Relocating Assembler (UniFlex-09 host)	$250	$65/$80
M09-FLX09	6809 Macro Relocating Assembler (Flex-09 host)	$250	$65/$80
M09-OS909	6809 Macro Relocating Assembler (0S9-09 host)	$250	$65/$80
M09-UNX11	6809 Macro Relocating Cross-Assembler (PDP-11/UNIX host)	$1200	$250/$300
M09-RSX11	6809 Macro Relocating Cross-Assembler (PDP-11/RSX11M host)	$1200	$250/$300
M09-PCDOS	6809 Macro Relocating Cross-Assembler (IBM PC/PC DOS host)	$375	$100/$135
MAN-MO9	Manual only (Specify Macro Assembler type)	$35	NA
MANEX-MO9	Additional Manual (Specify Macro Assembler type)	$20	NA

....

ORDERING INFORMATION

Introl software is available on the following floppy disk formats:

UniFlex formats: 8" SSSD 77 track
Flex formats: 8" SSSD 77 track; 5" DSDD 40 track; 5" SSSD 35 track
0S9 formats: 8" SSSD 77 track; 5" DSDD 40 track; 5" SSSD 40 tratk
PDP-11/UNIX formats: 8" RX01 Tar; 8" RX02 Tar; 8" Tektronix Tar
PDP-11IRSX11M formats: 8" RT-11
IBM PC/PC DOS formats: 5" DSDD

All prices are F.O.B. Milwaukee, Wisconsin. U.S.A. Prices and product specifications are subject to change without notice. All orders must be prepaid in U.S. funds drawn on a U.S. bank or shipped C.O.D. VISA and Master Card accepted. End users in Wisconsin, please add applicable Wisconsin State Sales Taxes. All domestic orders should include $10.00 shipping and handling, $25.00 for all overseas orders

Prices shown are for single-CPU use licenses. Site licensing and OEM licensing is also available.

An Introl Binary Software license Agreement must be completed and returned to Introl Corporation prior to software delivery.

Trademarks: Introl-C is a registered trademark of Introl Corporation; UniFlex and Flex are trademarks of Technical Systems Consultants; 059 is a trademark of Microware Systems; UNIX is a trademark of Bell Laboratories; IBM PC is a trademark of International Business Machines; PDP-11, RSX11, and RT-11 are trademarks of Digital Equipment Corp.

....

INTROL CORPORATION
647 West Virginia Street *** Milwaukee, Wisconsin 53204
Telephone (414) 276-2937

3/1/84

INTROL-C/6809 STANDARD LIBRARIES

(Representative Support Functions Provided)

STA09 LIB.	UFXO9 LIB.	FLX09 LIB.	OS909 LIB.	FUNCTION	DESCRIPTION
	X			abs	integer absolute value
	X			access	determine accessability of file
	X			acct	turn accounting on/off
	X			alarm	send alarm signal after specified time
X	X	X	X	alloc	allocate memory
X	X	X	X	atof	convert string to float
X	X	X	X	atai	convert string to integer
X	X	X	X	atol	convert strin to long
	X			brk	change core allocation
		X		cforkf	fork off a program
		X		chain	chain a new executable module from a C program
	X			chdir	change default directory
	X			chmod	change file access permission
	X			chown	change the owner of a file
	X		X	close	close a file
X	X	X	X	cprep	prepare environment for a C program
			X	crc	cycle redundancy count
X	X		X	creat	create a file
	X	X	X	cstart	runtime preparation routine
	X			dup	duplicate an open file descriptor
	X			dup2	duplicate an open file descriptor
X	X	X	X	ecvt	float to string conversion
	X	X	X	execl	execute a program
	X			execv	execute a program
	X	X	X	exit	exit a program with file cleanup
	X	X	X	_exit	exit a program without file cleanup
X	X	X	X	_extend	extend float
	X	X	X	fclose	close file
X	X	X	X	fcvt	float to string conversion
	X			fdopen	open a file
	X		X	fflush	flush file buffer
X	X	X	X	fgets	read file into string
	X			fdopen	open a file
	X		X	fork	spawn a new process
X	X	X	X	fprintf	formatted output Conversion
X	X	X	X	fputs	write a string to a file
X	X	X	X	free	free memory
X	X	X	X	fscanf	formatted input conversion
	X			fseek	seek to position in a stream
	X			fstat	get file status of open file
	X			ftell	tell the current position in a file
	X			ftime	current time
X	X	X	X	getc	get next character from a file
X	X	X	X	getchar	get a character from the standard input
	X			getegid	get effective group user id
	X			geteuid	get effective tiser id
	X			getgid	get group user id
	X		X	getpid	get process id
X	X	X	X	gets	read input into string
			X	getstat	get status of file or device id
	X		X	getuid	get user
	X			gtty	get status of file or device
X	X	X	X	index	find first occurrence of character
			X	intercept	intercept signals
	X			ioctl	control device
X	X	X	X	isalpha	test for alpha character
	X			isatty	test for terminal
X	X	X	X	isdigit	test for digit
X	X	X	X	islower	test for lower case
X	X	X	X	isspace	test for white space
X	X	X	X	isupper	test for upper case
X	X	X	X	itoa	convert integer to ASCII string
	X			kill	send a signal to a process
	X			link	link to a file
X	X	X	X	longjmp	non-local goto
	X		X	lseek	seek to a position in a file
X	X	X	X	malloc	allocate memory
X	X	X	X	max	return the maximum of two values
X	X	X	X	min	return the minimum of two values

INTROL-C/6809 STANDARD LIBRARIES (cont'd)

STA09 LIB.	UFXO9 LIB.	FLX09 LIB.	OS909 LIB.	FUNCTION	DESCRIPTION
	X			mknod	make special file or directory
X	X	X	X	modf	return fractional part of a float
	X			mount	mount a file sub-system
X	X	X	X	movmem	copy a block of memory from one location to another
	X			nice	change program priority
	X		X	open	open a file
	X			pause	stop until signal
	X			perror	print error message
	X			pipe	create an inter-process channel
X	X	X	X	printf	formatted output conversion
	X			profil	profile a process
	X			ptrace	process trace
X	X	X	X	putc	write a character to a file
X	X	X	X	putchar	write a character to the standard input
X	X	X	X	puts	write a string to standard output
	X		X	read	read from a file
			X	readln	read a line from a file
X	X	X	X	reverse	reverse a string in place
	X	X	X	rewind	reset specified file to beginning
X	X	X	X	rindex	find last occurrence of character
X	X	X	X	sbrk	allocate memory
X	X	X	X	scanf	formatted input conversion
			X	send	send a signal to a process
	X			setgid	set group user id
X	X	X	X	setjmp	non-local goto
			X	setstat	set status of file or device
	X			setuid	set the effective user id
	X			signal	catch or ignore signals
	X		X	sleep	suspend execution of process
X	X	X	X	sprintf	formatted output conversion
X	X	X	X	sscanf	formatted string conversion
	X			stat	get file status
	X			stime	set the system time and date
X	X	X	X	strcat	copy string
X	X	X	X	strcmp	compare strings lexicographically
X	X	X	X	strcpy	copy string
X	X	X	X	strlen	return string length
X	X	X	X	strncat	copy string
X	X	X	X	strncnp	compare strings lexicographically
X	X	X	X	strncpy	copy string
X	X	X	X	strsave	save string in memory
	X			stty	get status of file or device
	X			sync	update all disks
	X			syscall	C system call interface
	X			system	run a command string
	X		X	tell	return the current file position
	X		X	time	return the system time
	X			times	get process times
X	X	X	X	tolower	convert to lower case
X	X	X	X	toupper	convert to upper case
X	X	X	X	uldiv	unsigned long integer divide
X	X	X	X	ulmod	unsigned long modulo operation
X	X	X	X	ulmul	unsigned long multiply
	X			umask	set the default file access bits
	X			umount	unmount a file sub-system
X	X	X	X	_unext	unextend float
	X	X	X	ungetc	push character back on input stream
	X	X	X	unlink	delete file
	X	X		wait	wait for child process to terminate
	X	X	X	write	write to a file
			X	writeln	write a line to a file

NOTE: The STA09 Library is included with cross-compiler packages and with HLD cross-loader packages; the UNXO9 Library with UC6809 compilers and with ULD cross-loader packages; the FLX09 Library with FC6809 compilers and with FLD cross-loader packages; the OS909 Library with 0C6809 compilers and with OLD cross-loader packages.

Introl Corporation * 647 W. Virginia St. * Milwaukee, Wisconsin 53204 U.S.A.

Telephone (414) 276-2937

INTROL-C BINARY SOFTWARE LICENSE AGREEMENT
(Without Maintenance Option)

Introl Corp, (hereinafter called Licensor), for and in consideration of the terms and conditions set forth herein, and for a one-time license fee, hereby grants to Licensee, and Licensee accepts a personal, non-exclusive, non-transferrable license to use the binary software programs named below (hereinafter referred to as Licensed Programs) subject to the following terms and conditions:

DEFINITIONS: "Developed Programs" means any compiled or assembled program created by Licensee through use of the Licensed Programs, including the object code generated by the Runtime Library which is supplied as part of the Licensed Programs.

LICENSE: The Licensed Programs are supplied by Licensor solely for Licensee's internal business use on a single Designated CPU, identified below. This use includes the right for Licensee to construct Developed Programs using the Licensed Programs, and to sell, give away, or otherwise distribute the object code generated by the Runtime Library in creating these Developed Proarams. Except as provided in the preceding sentence, all right, title, and interest in and to the Licensed Programs and all related materials, including all source code furnished by Licensor with the Licensed Programs, remains the sole and exclusive property of Licensor. Neither this Agreement, the Licensed Programs, or any portions thereof, may be sold, leased, assigned, sub-licensed, or otherwise transferred by Licensee, except as expressly provided herein, without prior written consent of Licensor.

TERM: This License shall begin on the date hereof and shall remain in effect only as long and during such period as Licensee complies with the terms and conditions specified in this Agreement. This License Agreement may be terminated by Licensor if Licensee fails to comply with any terms or conditions specified herein. This License Agreement shall automatically terminate upon any act of bankruptcy by or against Licensee, upon any assignment for the benefit, of creditors of the Licensee, upon any attachment, execution of judgement or process against Licensee or its assets, or upon dissolution of Licensee.

LIMITED PERMISSION TO COPY LICENSED PROGRAMS: Licensee shall not copy, in whole or in part, any Licensed Programs which are provided by Licensor in machine readable form except for use by Licensee on the Designated CPU or for backup or archival purposes. This applies to copies in any form and generated by any means. Licensee shall maintain appropriate records of the number and location of all copies of the Licensed Programs, or portions thereof, and shall make these records available to Licensor upon request thereof. The original and any copy of the Licensed Programs, in whole or in part, shall at all times be the sole and exclusive property of Licensor. Licensee shall reproduce the following copyright notice on all copies of the Licensed Programs, in whole or in part, in any form: "Copyright 1983 by Introl Corp. Reproduction or publication in any form prohibited. Property of Introl Corp.". Use of the copyright notice is not to imply that the Licensed Programs have been published.

PROTECTION AND SECURITY: Licensee small not cause or permit disclosure of any Licensed Programs, in whole or in part, in any form, to any person other than Licensee's or Licensor's employees without prior written consent of Licensor. Licensee shall take all reasonable steps to safeguard the Licensed Programs so as to ensure that no unauthorized person has access to them, and that no unauthorized copies, in whole or in part, in any form, shall be made. Licensee expressly acknowledges that the Licensed Programs are confidential and proprietary property of Licensor and agrees to receive and maintain same as a confidential disclosure. Licensee further expressly acknowledges that unauthorized copying, use, or disclosure of the Licensed programs, in whole or in part, in any form, does great damage to Licensor and seriously impairs Licensor's ability to do business.

TERMINATION: Within thirty (30) days of termination of this Agreement for any reason, Licensee shall, at Licensee's option, either (a) return to Licensor all existing copies, in whole or in part, and their related materials, or (b) furnish to Licensor evidence satisfactory to Licensor that the original and all copies of the Licensed Programs, in whole or in part and in any form, have been destroyed.

DISCLAIMER OF WARRANTY: Licensor makes no warranties with respect to the licensed Programs. The licensed Programs are licensed 'as is' by Licensor, without warranty, and Licensor shall have no liability or responsibility to Licensee or any other person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by the Licensed Programs.

LIMITATION OF LIABILITY: THE FOREGOING WARRANTY IS IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. LICENSEE FURTHER AGREES THAT LICENSOR SHALL NOT BE LIABLE FOR ANY LOST PROFITS, OR FOR ANY CLAIM OR DEMAND AGAINST LICENSEE BY ANY OTHER PARTY, EXCEPT AS PROVIDED HEREIN. IN NO EVENT SHALL LICENSOR BE LIABLE FOR CONSEQUENTIAL DAMAGES, EVEN IF LICENSOR HAS BEEN ADVISED OF, THE POSSIBILITY OF SUCH DAMAGES.

MISCELLANEOUS: This Agreement constitutes the entire agreement between Licensor and Licensee and supersedes all prior agreements and representations. Licensee agrees to hold Licensor harmless on all liability associated with Licensee's breach of this Agreement including, but not limited to, all reasonable attorney's fees. This Agreement shall be governed by the laws of the State of Wisconsin in the United States of America and Licensee expressly submits to jurisdiction therein by process served by mail on Licensee at its below business address. Licensee agrees to advise Licensor of all changes in Licensee's address. Licensor's main address is given below. If any provisions of this Agreement, or portions thereof, are invalid under any applicable statute or rule of law, they are to that extent deemed to be omitted. The signing of this Agreement constitutes acceptance of the terms of this Agreement. No provision in correspondence or on Purchase Orders shall in any way modify this Agreement. Licensor represents that it has sufficient right, title, and interest in and to the Licensed Programs to make this Agreement with Licensee.

Licensee Name:____

Normal Business Address of Licensee:

____

____

____

____

Country: ____

Phone: ____

Licensed Program Name: _________________________________ S/N __________________

Purchased From: ____

DESIGNATED CPU:

Mfgr: _________________________ Model # _____________ S/N __________________

LICENSEE SIGNATURE: ____

Name And Title: ____

Date: ____

* *

INTROL CORP. * 647 West Virginia Street * Milwaukee, Wisconsin 53204 U.S.A
Telephone: (414) 276-2937

*********** ICC COMPILER **********

-a[t|d|b|s)=(loc) - Place 'text', 'data', 'bss', or 'string' type data under (loc) location counter
-b=(directory) - Find current and subsequent C Compiler passes in (directory) location.
-c - Override default condition with respect to generation of position dependent/independent code.
-d - Override default condition with respect to generation of position dependent/independent data.
-g(c) - use alternate version of preprocessor pass for compilation (RC6809 and 0C6809 compilers only).
-i=(directory) - Search (directory) location for #include files.
-k - Display progress of compilation/assembly sequence on console.
-m(name)[=(string)] - Define (name) in preprocessor, with value (string) optionally assigned to (name).
-n - Inhibit execution of next compiler pass in the compilation sequence.
-r - Save C Compiler's intermediate assembly language output file.
-s - Disallow nested comments.
-s=(size) - Set maximum size of triple buffer.
-t=(directory) - Place C Compiler's temporary files in (directory) location.
-y=[=(n)] - Strip all identifiers to a maximum length of (n) characters.
- z Interpret "\n" (newline) characters as being carriage returns.

*********** R09 RELOCATING ASSEMBLER ***********

-a - Place all symbols except those beginning with a "?' character in the object file.
-c - Send Assembler's output listing to console.
-i - Include all included files in output listing.
-j - Include symbols beginning with a "?" character in the symbol table listing.
-l=(filename) - Place output listing in specified file.
-n - Do not produce an output listing.
-o=(filename)- Assign name to Assembler's relocatable output module.
-q=(class) - Assign numeric class identifier (0 through 255) to relocatable output module.
-s - Suppress listing of the symbol table.
-u - Force all undefined symbols to default to imported symbols.
-x - Don't generate an object module.
-z - Delete input file when Assembler has finished using it.

************* ILINK LINKER ******************

-b- Do not search the default Standard Library.
-c=(file) - Get additional link-time parameters from command file.
-d[(c)] - Call optional cross-loader named "(c)LD" when Linker finishes.
-e=(symbol) - Set entry point.
-f(string) - Search additional Standard Library named "lib(string).R"
-l[s][x][u][=(file)] - Produce a linker output listing.
-m=(symbol) - Define primary function naming symbol.
-n - Inhibit Linker from automatically calling Loader.
-o=(file) - Assign name to output file.
-p[(c)] - Pipe Linker's output to loader.
-r - Save Linker's output file (during automatic link-and-load operations).
-s - Strip output file of all non-entry-defined symbols.
-t=(classlist) - Use (classlist) classes of modules during linking process, if they are available.

************* HLD LOADER *************

-a=(seg);(placernent)[,(seg);(placement)] - Set segment memory bound (segment may begin, or end, at a specific memory location, or specified to immediately follow, or immediately precede, another segment).
-c=(file) - Get additional parameters from command file.
-g=(type) - Set output format (Motorola S Record, Intel Hex, Tek Hex, or Extendend Tek hex format).
-h - Define EOL character to be carriage return (rather than newline character).
-l[s][=(file)] - Produce a Loader output listing.
-o=(name) - Assign name to output file.
-u=(seq) - Place uninitialized data in specified segment.
-v[(char)] - Modify Loader's symbol changing procedures for symbols beginning with non-alpha characters.
-w - Produce executable output file no nutter what.
-x (type):(ext) - Set output filename extension for specified type of hex output format.
-z - Delete loader's input file when Loader has finished using it.

************* ULD LOADER ************

-a=(sec):(seg)[,(seg)] - Assign location counter segment to UniFlex program section (text, data. or bss).

-c=(file) - Get additional parameters from command file.

-l[s][=(file)] - Generate loader output listing.

-o-(name) - Assign name to output file.

-v=(size) - Set stack section size.

-w - Produce an executable output file no matter what.

-x[=(pagesize)] - Produce output file in UniFlex segmented format.

-y=(origin) - Set text section origin at specified location.

-z - Delete Loader's input file when Loader has finished using it.

************* FLD LOADER **************

-a=(sec):(Seg)[,(seg)] - Assign a location counter segment to a Flex program section (text, data, or bss).

-c=(file) - Get additional parameters from command file.

-l[S][=(file)] - Produce an output listing.

-o=(name) - Assign name to output file.

-w - Generate executable output file no matter what.

-y[t|d|b|]=(origin) - Set origin for text, data, or unitialized section of output file.

-z - Delete input file after Loader has finished using it.

*********** OLD LOADER ************

-a=(sec):(seg)[,(seg)] - Assign a location counter segment to an OS9 program section (text, data, or bss)

-c=(file) - Get additional parameters from command file.

-l[s][=(file)] - Produce an output listing.

-o=(name) - Assign name to output file.

-V=(size) - Set stack section size.

-w - Generate an executable output file no matter what.

-x - Place executable program module and data initialization information module in separate files.

-z - Delete the input file after the Loader has finished using it.

************ LIBMAN LIBRARY MANAGER COMMANDS ************

a (file),(module)[,(class)] - Add module to library; create new library.

d (nodule)[,(class)] - Delete module from library.

r (file),(module)[,(class)] - Replace module in library.

q - Quit Library Manager 7after saving library file being edited).

omit - Exit Library Manager (without saving edited file).

l (module)[,(class)] - List information on named file.

sl (module)[,(class)] - List abbreviated information on named file.

h - Provide on-line help.

lo (file) - Explicitly load a library file.

ll (file) - List a loaded library.

sll (file) - Provide abbreviated listing of a loaded library.

s (file) - Save library using the filename indicated by (file).

c (file) - Get additional commands from named command file.

e (strings) - Echo specified strings to the terminal.

f (module)[,(class)] - Find named module.

p (module)[,(class)] - Print information for named module.

sp (module)[,(class)] - Print abbreviated listing of information for named module.

i (file),(module)[,(Class)] - Insert named module in library so it precedes current module.

INTROL CORPORATION

647 West Virginia Street ** Milwaukee, Wisconsin 53204 U.S.A.

Telephone (414) 276-2937