INTROL-C COMPILER

REFERENCE MANUAL

The contents of this manual have been carefully reviewed and are believed to be entirely correct. However, Introl Corp. assumes no responsibility for any inaccuracies.

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 Program	Default Filename Input Files	Extension 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.

<u>INPUT FILE SPECIFICATION</u>
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.

<u>INPUT LINE</u>
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
```

<u>SYMBOLS</u>
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.

dc - Define Data Constant

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.

ds - Define Data Storage

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).

end - End of Assembly

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.

equ - Equate Svmbal With A Value

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
```

err - Programmer-Generated Error

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.

export - External Symbol Definition

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.

fcb - Form Constant Byte

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.

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

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.

char to float: 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.

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

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

char to short int: Characters are converted to short by padding zeros on the left.

char to unsigned int: Characters are converted to unsigned by padding zeros on the left.

char to pointer: Characters are converted to pointer by padding zeros on the left.

float to char: 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.

float to 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 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.

float to long 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 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

I.1