INTROL

LINKER AND LOADER
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 inaccuracies.

The software described in this manual is proprietary and is furnished under a license agreement from Introl Corp. The software and supporting documentation may be used and/or copied only in accordance with said license agreement.

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

tel. (414) 276-2937

Table of Contents

Linker And Loader Reference Manual

LINKER

The function of the Linker is to join several relocatable object modules together to form a single relocatable object module as the result. Normally, when the Introl Linker finishes, it will automatically call the Loader, causing the object module produced by the Linker to be then translated into an executable file by the Loader. Once such executable file has been generated, the actual object module generated by the Linker is normally automatically deleted. Thus, although the Linker itself produces an intermediate relocatable module, the more usual result of a linker command line call is an executable file that is subsequently produced by the Loader. Options are provided, however, to permit the Linker's output module to be retained even though an executable file has been produced; also, an option exists to inhibit the Loader call entirely when the desired result is simply the relocatable module generated by the Linker.

LINKER COMMAND LINE

The general form of the link command line is:

ilink <files> {<options>} {<files>} {<options>}

where <options> can be zero or more Linker and Loader option specifiers (described later in this Section), and <files> are the filenames of the relocatable files or libraries which are to be input to the Linker. Unless an option to inhibit loading is explicitly specified on the command line (the "-n" option), the Loader will be automatically executed when the Linker finishes.

The Linker expects each of its input files to have a filename extension; if none is explicitly defined, the filename extension is assumed to be ".R", which is the filename extension normally assigned to relocatable files generated by the Assembler. If the Linker is being run independently (ie with the "-n" option specified, which inhibits the automatic call to the Loader), the Linker will produce a relocatable module as the end result, having the filename extension ".RL". Such modules (ie modules which have been linked but not loaded) are themselves relocatable modules which can be legally reused as inputs to the Linker, if desired. If the Loader call is not explicitly inhibited, a link command line call will result in generation of an executable output file as the final result (ie, the file produced by the Loader pass). In this latter case, the intermediate relocatable module generated by the Linker (ie the file having a ".RL" filename extension) will not be retained unless the user specifically opts to do so (via the "-r" Linker option). In either case, the filename assigned to the output module(s) produced as a result of the linker call will be determined by the "primary function name" symbol, which is discussed under Operation, below.

OPERATION

When the Linker is first invoked, it begins its linking process by attempting to resolve two references which are implicit to the Linker. The first is called the "primary function name", the second is the program "entry point". The user may, as an option specification on the link command line (the "-m=<file>" option), specify any symbol as a primary function name. If none is explicitly defined, however, the primary function naming symbol will be assumed to be "_main", the symbol that represents the name of the usual starting function ("main") in a C program. The filename of the module in which the Linker finds the primary function name will normally be the name assigned to the Linker's relocatable output module, but with the filename extension ".RL" being appended to the Linker's output module.

The Linker begins its search by first searching through all of the files specified on the link command line, searching these files in the order they are listed, attempting to resolve the primary function name. If it succeeds, it will include the module which contains the definition of the primary function name, and will then proceed to resolve any external references which that module makes. (If the primary function name cannot be found, the Linker automatically loads the Standard Library and attempts to resolve the "entry point" symbol, as described below.) When all possible external references caused by inclusion of the module containing the primary function name have been satisfied, the Linker will then attempt to resolve the "entry point" symbol. In doing so, the Linker will first search through the files on the link command line, and then search the Standard Library if necessary, looking for a module which has an entry point symbol defined. If it finds one, it will include the module which contains the entry point and attempt to resolve any resultant external references that module makes.

An unmodified Standard Library will always contain a module for which an entry point is defined. This is the module usually used to set up the environment required before the first C function (usually "main") can be executed. The Compiler itself does not normally define an entry point when it produces a module. An assembly language programmer, however, may specify the entry point of an assembly language module by placing the name of the entry point following the END assembler directive. If there is more than one module with an entry point defined, the Linker will assume the entry point is that of the first such module it finds after beginning its search. It begins its search with the files on the link command line, scanning left to right, and then searches the Standard Library, top to bottom. Therefore, if a module on the link command line defines an entry point, that module will be the first module found by the Linker and, therefore, will be the one selected for inclusion (ie rather than the module contained in the Standard Library). If no module on the link command line contains an entry point, the Linker will assume the entry point symbol is "cstart", which happens to be the usual name for the Standard Library routine which sets up the environment for a C program.

The Linker terminates when it has no more external references to resolve or, alternatively, when it runs out of files to search in attempting to satisfy any unresolved references that might still exist. The Linker's output will be a relocatable module that has the same name as the name of the module which contains the primary function name, but with a ".RL" filename extension appended. When the Linker has determined it has resolved all the external references it possibly can, it will automatically call the Loader. If all external references have been successfully resolved by the Linker, the Loader will load the Linker's output into an executable output file. If unresolved references still exist, however, the Loader will complain and loading of the module will be unsuccessful.

As indicated above, it is perfectly legal to use the Linker to link several modules together which, of themselves, do not satisfy all the external references they make. This feature is very useful when it is desired to link two or more relocatable files together to produce a single resultant "partially linked" module (which may contain some unresolved references). Such partially linked modules may themselves then be reused as inputs in subsequent linking operations, and linked with other relocatable modules as necessary. In such-cases, when it is the user's intention to do partial linking of this type, a user option ("-n") to prevent automatic execution of the Loader must be specified on the link command line.

In many cases, such as for a compiled C program contained in a single module, calling the Linker may be as simple as specifing the name of a single relocatable file produced by the Compiler. For example, if the file to be linked and loaded had the name "test.R" (which is the file that would be produced by the Compiler if the user had compiled a program called "test.c"), the user could call the Linker by entering the following:

```
ilink test
```

For this example, the Linker would proceed to first link the file 'test.R" with applicable referenced functions from the Standard Library ("libc.R"), producing the linked module "test.RL" as an intermediate result. It would then automatically call the Loader, which would load "test.RL" into either an executable file or a file of load records, as appropriate to the type of Introl Loader being used. Since the "-r" option was not specified on the linker command line for this particular example, the Loader would also automatically delete the "test.RL" file when it had finished using it. Note that it is unnecessary to specify the Standard Library, "libc.R", on the command line; the Standard Library is always implicit to the Linker when it is called.

LINKER CLASS LIST

Each relocatable module produced by the Assembler, as well as each module contained in the Standard Library, has an attribute called

its "class", which is a user-assignable number from "0" (zero) to
"255". During the linking process, the Linker always uses the module's class number in combination with the module's filename for module identification purposes. The class number is, in effect, an "extra identifier" that provides a mechanism for distinguishing between several identically named modules that may be contained in a library.

The default "class" for modules produced bv the Assembler is "0"; however, any other legal class number (ie "1" through "255") may be selectively assigned to any of these modules by the user. Similarly, most of the library routines contained in the Standard Library, libc.R, have a preassigned class number of "C", although several non-zero class modules are also supplied. For example, libc.R contains 3 different classes of the ofmt routines used by the "printf", "fprintf", and "sprintf" Standard Library functions (classes 0, 5, and 6) and 3 different classes of the imft routines used by the "scanf", "fscanf", and "sscanf" Standard Library functions (classes 0, 7, and 8). The class 0 ofmt routine supports longs, ints, and floats; the class 5 ofmt routine supports longs only; and the class 6 ofmt routine supports longs and ints. Similarly, the class 0 ifnt routine supports longs, ints, and floats; the class 7 ifmt routine supports only longs; and the class 8 ifmt routine supports longs and ints.

Because of a relocatable module's class attribute, one of the link time options available to the user is the specification of a "linker class list" on the link command line. Use of a class list specification is only necessary when the user wants modules other than class "O" modules to be considered for inclusion by the Linker.

The linker class list specification defines two things to the Linker: (1) it defines the specific non-zero classes of modules that should be potentially considered for that particular link process, and (2) it simultaneously establishes a priority ranking of these classes of modules, which enables the Linker to choose the "correct" module from among possibly several that may have been given identical filenames in a library.

A linker class list is specified on the link command line as one or more <option> entries of the form:

t=<class list>

where <class list> is a series of one or more numerical values from "1" through "255" (see -t option below). The numerical values contained in <class list> represent those specific non-zero module classes, listed in the order in which they are to be "preferred" for possible use, which are to be considered potentially valid for inclusion for that particular link process. Modules of class "O" are ALWAYS implicit in any class list specification and therefore are not included in a linker class list on the command line. The Linker automatically assigns lowest "preference" to class "O" modules and will only use a class 0 module if it cannot find some other

L.1.4

identically named module having one of the non-zero classes defined
in the linker class list.

As mentioned earlier, a class list specification on the linker command link is only necessary if modules having a class other than "0" are to be considered for use by the Linker. When a class list is specified, however, it is important to note that the order in which any class numbers appear on the command line is just as significant to the Linker as the actual class numbers themselves. This is because the Linker (which scans the entire command line from left to right to determine all of the acceptable classes) assumes that the class numbers are listed by the user in ordered sequence on the command line, with the "most preferred"'class being the class it first encounters on the command line, the "next most preferred" class being the second class it encounters, and so on. The Linker will always select the "most preferred" class of any given named module that it can find.

An ordered class list of this type is necessary for the user to unambiguously define, and the Linker to properly select, the intended module in many instances. For example, suppose the user had compiled and assembled a program module, "file1", (with a class of "0") that referenced two library routines contained in the Standard Library, one called "abc" and the second called "xyz". Further assume that two different versions of the abc module existed, one with class 0 and the other with class 1; and three versions of xyz existed, one with class 0, one with class 1, and one with class 2. If the user wanted to link file1 with the class 1 module of abc and the class 2 module of xyz, he could enter a link command line such as:

ilink file1 t=2,1

In this case the Linker would ascertain that, given the choice, it should give highest preference to using class 2 modules, next highest preference to class 1 modules, and lowest preference to class 0 modules. During the linking process the Linker would first look for a class 2 filel module and, failing that, then look for a class 1 file1 module and, failing that, then look for a class 0 file 1 module, which it would find and therefore include. The Linker would then begin searching the Standard Library to resolve the references filel makes to abc and xyz. it would begin its search for abc by first looking for an abc class 2 module and, failing that, then begin looking for an abc class 1 module which it would find and link in with filel to resolve the reference made to abc. Similarly, it would begin its search for xyz by first looking for an xyz class 2 module which it will find and link in to file1 to resolve the reference made to xyz. Aithough an abc class 0 module and xyz class 1 and xyz class 0 modules also existed in the library, these would have been ignored by the Linker inasmuch as it had been able to find "more preferred" versions of abc and xyz.

By comparison, if the user had used a link command line such as

L.1.5

ilink t=1,2 file1

the Linker would instead have given highest preference to class 1 modules and next highest preference to class 2 modules, with class 0 modules again having lowest priority (as is ALWAYS the case for class 0 modules). In this case the Linker would first look for a class 1 file1 module, then a class 2 file1 module, and then a class 0 file1 module which it would find and include. The Linker would then look for, find, and link in the ("most preferred") class 1 abc module; then look for, find, and link in the ("most preferred") class 1 xyz module. The class 2 xyz module would ONLY have been considered for inclusion in this instance if the Linker were unable to find the "more preferred" class 1 module, which of course it does find in the example situation given.

Notice that the class list may contain multiple class specifiers and that class zero is ALWAYS implicit in any class list specification.

LINK COMMAND LINE OPTIONS

Linker options, as well as Loader options, may be specified on the link command line. Loader options, if specified, will be passed on to the Loader when it is automatically called by the Linker. The "linker-specific" options listed below are those options which apply specifically to the Linker, per se. The Loader options that may also be specified on the link command line are discussed in the Loader Appendices to this manual.

Linker-Specific options include:

-b

This option prevents the Standard Library, "libc.R", from being searched by the Linker. Usually this option is specified in combination with the "-f" Linker option, discussed below, when programs are being

-c=<file>

The option specifies that <file> is a command file where the Linker will find additional information. The command file is a text file which may contain extra options and additional file names to be referenced following those listed on the command line. Each option or file name must appear on a separate line in the command file.

-d[<c>]

This option is used for specifying, at link time, which of several (optionally available) Introl Loaders is to be called by the Linker when linking is completed. Specifically, use of this option will cause the Linker to call the Loader whose Introl filename is "<c>ld", where the <c> represents the first character of the desired Loader's "name". For example, the

option specification "-dh" would instruct the Linker to call the Loader named "hld" when it finishes (assuming of course that the "hld" Introl Loader is actually available for use). If the -d[<c>] option is not specified, or if there is no character specified via the <c> entry, the Loader selected for use will default to the "standard" Loader supplied with the Compiler. (In general, the "standard" Loader is one which produces code that is executable on the Compiler's host operating system.) The several different types of Loaders that are optionally available for use, and the "<c>ld" names associated with each, are described in the Loader Appendix of this manual.

NOTE: When an "optional" target- system- dependent-type of Loader is being specified for use, the compatible "standard library" supplied with that optional Loader must also be specified for use during the linking process. In such cases the "-b" Linker option can be used to inhibit the Linker's use of the "standard" libc.R library, and the "-f" option used to instruct the Linker to instead find and use the "optional" standard library which is compatible with the target operating system.

-e=<symbol>

This option sets the entry point. If the <symbol> being specifed as the entry point refers to a C symbol that has been generated by the Compiler, the <symbol> name must include a leading underscore character (ie the Compiler automatically pre-pends a leading underscore to all symbols it generates). If this option is not used, the Linker will search through all the modules in the order they are listed on the command line, and then search the Standard Library if necessary, in an attempt to find one which has an entry point defined. The entry point will be that of the first such file the Linker finds. If no input module specifies an entry point, the Linker will usually find one called "cstart" in a module of the same name in the Standard Library. For assembly language programs, an entry point is placed in a module by placing the desired entry point symbol on the "end" directive in an assembly language file (see Assembler section of the Compiler Reference Manual).

-f<string> or -f=<string>

This option, which has two forms, is used to specify that additional libraries will be found in the standard library place which are to be searched by the Linker (ie libraries that are to be searched in addition to the Standard Library, libc. R) .The "-f<string>" form of the option specifies that an additional library to be searched is named "lib<string>.R", where <string> represents any series of characters. The "-f=<string>" form specifies that an additional library to be searched is named "<string>.R", where <string> can represent any string of characters. This option must normally be used (together with the "-b" option mentioned above) when an "optional" Loader is being called; this is necessary so that

the Linker uses a "standard library" which is compatible with that particular Loader.

-l[s][x][u][=<file>]

This option causes a linker listing to be produced. The optional file name indicates that the listing is to be placed in the indicated file rather than being listed on the console. The "s", "x" and "u" characters are all optional and affect the listing's contents, as follows: If the "s" character is specified the listing will include all symbols. If the "X" character is specified the listing will include a cross reference symbol listing. If the "u" character is specified the listing will include a list of the modules taken from each the files specified on the command line. Any combination of these three characters may be specified.

-m=<symbol>

This option defines the primary function naming symbol. The primary function name is the external reference which the Linker attempts to resolve first. If left unspecified, the naming symbol defaults to "_main", which is usually the primary function in a C program. (At the C program level this primary function name is specified as "main", but the leading underscore is added by the Compiler, as is the case for all symbols generated by the Compiler. It is therefore important to remember that, when specifying a naming symbol that is contained in a compiled module, the symbol will always begin with a leading underscore.) The filename of the module which contains the primary function name is normally the name that will be assigned to any file(s) produced as a result of a Linker call line.

-n

This option prevents the Loader from being automatically executed when the Linker finishes. When the "-n" option is not specified, the Linker will normally default to calling the "standard" Loader (unless some other loader type has been optionally specified using the "-d(<c>]" option discussed previously).

-o=<file>

This option is used to assign a specific name, represented by <file>, to the Linker's output file. If this option is not used the output file will be given the same name as the module in which the primary function name is found. If no filename extension is explicitly specified, the Linker output filename will default to having a ".RL" extension.

-P[<C>]

This option is useful only an Unix-like operating systems, such as UNIX, INIX, and TNIX for example. On such systems, it causes the output of the Linker to be piped to the Loader rather than to be transferred in a temporary file. On some systems this

will cause a noticeable speed, improvement. The [<c>] indicates an optional character which may be used to specify that the Linker output should be sent to a particular optional Loader when use of the default "standard" Loader is not desired. The <c> character, when specified, represents the first letter in the Introl name of the desired Loader, just as for the case of the "-d[<c>]" option described previously.

-s

This option specifies that the output file is to be stripped of all non-entry defined symbols. This is useful when producing a partially linked module in which the user wishes to "hide" all the already resolved symbols. Partially linked modules are typically modules that have been linked, but not loaded, which may still contain unresolved references.

-t=<classlist>

This option is used to define an ordered listing of those non-zero class numbers, between 1 and 255, which are to be "preferred" for use in the linking process. The <classlist> can be a series of one or more numbers from "1" through "255". When a class list contains multiple class number entries, a comma or period must separate successive class numbers, as in "t=3,7,4", for example, which specifies the classes "3", "7", and "4". The order in which class numbers are entered on the link command line is significant to the Linker and defines the order of class preference. The first-entered (ie left-most) class appearing on the link command line will be given highest preference for inclusion by the Linker, the second-entered class will be given next highest preference, and so on. Modules of class 0 are always considered by the Linker as having lowest priority and are used in the linking process only if an identically named module having a class number which is included in the linker class list specification cannot be found by the Linker. For example, a class list such as "t=3,7,4" tells the Linker to preferably use modules of class 3 (if they can be found), or else use class 7 modules (if they can be found), or else use class 4 modules (if they can be found), or else, as a last resort, use modules of class 0 (if they can be found).

The reader is referred to the Loader Appendices of this manual for applicable Loader options that may be specified on the link command line.

It is the Loader's function to fix absolute addresses for the relocated values in a relocatable module, thereby converting a relocatable module into an "executable" output file. The Loader is usually called automatically by the Linker but it may also be called separately by the user. As indicated below, several different Loaders are (optionally) available for use with Introl-C and, if the user has elected to obtain such optional Loaders, a variety of executable output file formats can be generated, depending on the Loader being used.

Each resident Introl-C compiler package, and each Introl-C cross-compiler package, nominally includes a single, specific type of Introl Loader which is considered as being the "standard" Loader for that compiler's particular host system configuration. For resident Introl-C Compiler packages, the 'standard' Loader that is furnished is an "operating system dependent" type of Loader which generates an output file that is executable on that particular Compiler's host system. For cross-compiler versions of Introl-C, the "standard" Loader furnished is typically a "hex" type Loader that generates a file of output load records, which can be either Motorola S-Records, intel Hex, Tektronix Hex, or Tektronix Extended Hex at user option. Besides the "standard" Loader that accompanies any given Compiler type, it is also possible for the user to optionally obtain and use other compatible "cross-Loaders" which generate output formats unrelated to the Compiler's host operating system. For example, "hex-type" Loaders are optionally available for use with resident versions of Introl-C; "operating system dependent" type Loaders are optionally available for use with cross-compiler versions; etc.

There are, therefore, several different species of Loaders, (as well as several different types of related Standard Libraries) that may potentially be used under Introl-C. The "standard" Loader supplied with your Introl-C package, as well as any other Loaders that may have been optionally ordered, are described in detail in the Loader Appendix of this Linker Reference Manual. This Loader section describes the general features that are common to all Loader types.

Normally the input to the Loader is expected to be a relocatable file which has no unresolved external references; if unresolved references do exist in its input, loading will normally not be successful. A Loader option is provided, however, to force a file to be loaded even if it contains unresolved references.

Usually a relocatable file has to be linked before it can be used as input to the Loader. It is also possible, of course, to assemble a file which makes no external references and then use the relocatable output file produced by the Assembler directly as input to the Loader (ie without having actually linked it).

LOADER COMMAND LINE

The "standard" Loader supplied with your Introl-C package (see Loader Appendices to this manual) is normally automatically called by the Linker when the Linker pass finishes. However, linker command line options exist (see Linker Section of this manual) that mav be used to alternatively force the Linker to automatically call other optional Loaders (assuming such optional Loaders have been obtained for use). Situations also arise when it is desirable to explicitly call the Loader alone, without first executing the Linker. When such situations arise, the Loader may be independently called by the user with a loader command line of the general form:

<c>ld <file> {<option>}

where <c>ld represents the Introl filename of the specific Loader being called, <file> is the name of the (linked) relocatable module which is to be loaded, and (<option>) represents zero or more Loader option specifiers.

Each of the potentially usable Introl Loaders is uniquely identified by a 3-letter Loader filename, the last two letters of which are always "ld". The <c> designator indicated in the "<c>ld" loader call on the command line therefore represents the first letter in the three-letter Loader name. For example, to call the Introl hex type of Loader, which has the filename "hld", the "<c>ld" entry on the command line would actually become "hld". For further specifics on the names of the loaders which can be legally accessed, refer to the Loader Appendices of this manual.

The relocatable file that is input to the Loader is expected to have a filenarne extension; if none is specified, the default filename extension ".RL" is assumed. Normally the name of the executable output file will be identical to the name of the input file, but with a filename extension typically added by the Loader. The filename extensions each Loader appends are discussed in the Loader Appendices to this manual.

LOADER OPTIONS

Each type of Loader available for use with the Introl-C has its own, generally unique set of options. The specific options that apply to each Loader furnished are discussed in the Loader Appendices.

When the Loader is being called separately, Loader options are specified directly on the loader command line when the Loader is being automatically called by the Linker, Loader options are specified on the link command line, together with the Linker options. If Loader options are specified on the link command line, any such options (ie those that do not apply to the Linker) will be automatically sent on to the Loader. For the most part Linker and Loader option specifiers tend to be distinct, so that there is little ambiguity when Loader options are specified on the link command line.

LIBRARY MANAGER

This section describes the features and operation of the Introl Library Manager.

For a program to be succesfully linked and loaded, all its external references must be resolved. That is, any functions which are referenced by the program but not included in the program must be added to it at link time. The Linker can be directed to search various files to find already compiled functions which satisfy these references. When it finds a piece of compiled code which satisfies a reference it includes the code in the resultant program. Any compiled or assembled file may be a legitimate input to the Linker. To facilitate the Linking process, it is often useful to have a file which contains more than a single piece of compiled code so that the user can specify a whole series of routines to the Linker with a minimum of fuss. Such a file is called a library file, an example of which is the introl-C Standard Library (libc.R). The Linker can search a library file and selectively extract only those modules it requires to link the file.

LIBRARY FILES

A library file is a file which contains one or more linkable object modules of the type produced by the Introl Assembler. When a file is compiled and assembled, the result is exactly one linkable module which is placed into a file. This file is actually a library which happens to contain only a single module. When the user links a program, one or more of these "libraries" are specified on the link command line. Usually the "libraries" are those produced as a result of a compilation and contain only a single module, however, they may also contain several modules. The Library Manager, "libman", is a program which allows the user to place several modules into a single library file. When the user has a large set of modules which are commonly used in programs, it is usually convenient to place them all in one library and then simply specify the library once on the link command line. The Linker will extract only those modules it requires in order to satisfy the external references of the program.

The Linker is designed to automatically search the "Standard Library", libc.R, if it still has external references to satisfy after it has exhausted all the alternatives provided by the modules specified an the link command line. For many C programs, the Standard Library is usually where most of the external references are satisfied. Many users find it useful to add to, or modify routines in, the Standard Library.

The Library Manager is the utility program which allows the user to create new libraries and also to maintain existing ones.

LIBRARY MANAGER

Because any file that is produced by the Assembler is already technically a library file, the Library Manager can correctly be

looked upon as a program which manipulates libraries. Its input is a library file, such as a linkable object file produced by the Assembler. Thus, in the description below, references to "libraries" also implicitly includes those files output by the Assembler.

The Library Manager is called by entering a command line of the form:

libman <lib> {<optional-direct-command>}

where <lib> is the name of the library to be edited and <optional-direct-command> is an optional command to the Library Manager. If the <optional-direct-command> entry is omitted, the Library Manager will enter its "Interactive Mode" of operation and solicit library management commands from the user terminal.

The input library specified by <lib> may be either a new library or an existing one and, unless the user takes contrary action, it will also be the nane of the output library.

MODES OF OPERATION

The Library Manager has three modes of operation: Direct Mode, Interactive Mode, and Command File Mode. The most convenient to use for simple additions and deletions to the library is the Direct Mode. For more extensive modifications the user may instead wish to use Interactive mode. The third mode is the Command File mode which causes the Library Manager to read its commands from a file rather than getting them from the user terminal.

Direct Mode: In Direct Mode the user is permitted to specify a single command on the library manager command line. When the Library Manager is called, it executes this single command function and then immediately exits from the Library Manager. When modifying libraries, however, a single command function is often all that is necessary to accomplish the change desired by the user. When Direct Mode is being used, the desired command is specified right on the command line, following the <lib> library specification. Any Library Manager command may be used in the Direct Mode.

Interactive Mode: if no command is specified on the Library Manager call line, the Library Manager will enter its Interactive Mode of operation. In Interactive Mode the Library Manager will print a colon (":") as a promet and will accept a succession of commands directly from the user terminal. Interactive Mode is useful when the user must make extensive changes to a library, or when the user wishes to step through the library checking and/or changing modules in an "interactive" manner. Once selected, the Interactive Mode will remain in effect until the user enters a "quit" or "omit" command.

Command File Mode: One of the commands which the user can specify as an Interactive code or a Direct Mode command entry is the "Comfile" command. This command instructs the Library Manager to read subsequent instructions from a command file. When a "Comfile"

command is entered, the Library Manager will read from the file specified until it reads a "quit" or "omit" command or, alternatively, until it reaches the end of the file. when exiting the Command File Mode, the Library Manager will return to whatever mode it was in before the Command File Mode was entered. If the Command File Mode was entered as the result of a Direct Mode command, then the Library Manager will terminate when Command File Mode is exited. If entered from the Interactive Mode, it will return to the Interactive Mode.

LIBRARY MANAGER COMMANDS

In the descriptions that follow, the commands may be abbreviated to the characters shown in capital letters. For simplicity, the descriptions are specified in a BNF type form. In this form items enclosed in angle brackets "<" and ">" represent names or numbers to be chosen by the user. Items enclosed in square brackets "[" and "]" represent optional items. Anything enclosed in curly brackets "{" and "}" may be repeated zero or more times. These "meta" characters (ie <,>,{,},[, and]) are just to help the user understand what is required and should not actually be typed in. Thus the "delete" specification ...

Delete {<module>{,<class>])

means that the delete command (which may be abbreviated to just "d") requires zero or more user-specified module names, each of which may have an optional class specifer which is separated from the module name by a comma.

In the following:

<module> refers to the name of a module (which should consist of a series of characters). The first character may not be a digit.

<file> refers to any legal file or path name.

<class> is a number from 0 to 255 which represents a module's class number.

Thus a legal example of the delete command could be:

d modulea,2 moduleb modulec, 0

which would cause three modules to be deleted; the class 2 "modulea" module, the class 0 "moduleb" module, and the class 0 "modulec" module.

Add {<file>{,<module>f,<class>]}}

The add command is used to add modules to an existing library or to create a new library. It consists of the word "add", which may be abbreviated to "a", followed by one or more filenames, each of which may be followed by zero or more module specifications, each of which

may include a class specification. It is possible to add modules at a specific place in the library (see the "find", command) but for most linking applications it makes no difference where a module is located in the library. In Direct Mode, the add command will add modules to the end of a library. In Interactive Mode or Command File Mode, the Library Manager can be directed to add a module anywhere in a library. The argument to the add command is a filename which should contain at least one linkable module (such as that produced by a compilation). The filename may be followed by any number of module names. If there are no specifications following the file name, the Library Manager will attemct to add all of the modules contained in the file. If specific nodules are named, the Library Manager will attempt to add only those modules from the named file. Any module may have an optional class specification, which is a numeric specifier in the range of 0 to 255. If the class specification is not present, the first module encountered having the specified module name, regardless of its class, will be added to the library; otherwise only a module with a matching name and class will be added. The add command will not add any module whose name and class match one already existing in the library.

Delete {<module>{,<class>]}
This command allows the user to delete modules from a library. The delete command will attempt to delete the named modules, taking into account the module's class, if it is specified. If the class specifier is omitted, and there is more than one module having the specified name in the library, the delete command will print a warning message and will not delete the module. The user may then delete the module by specifying the class of the module which is to be deleted.

The delete command will print a warning message if no module name is specified.

Revlace {<file>{,<module>{,<class>]}}
The replace command is used to replace modules in an existing library. It consists of the word "replace", which may be abbreviated to "r" followed by one or more filenames, each of which may be followed by zero or more module specifications, each of which may include a class specification. The argument to the replace command is a file name which should contain at least one linkable module (such as that produced by a compilation). The filename may be followed by any number of module names. If there are no module specifications following the file name, the Library Manager will attempt to replace all of the modules contained in the file. If specific modules are named, the Library Manager will attempt to replace onlv those modules. Any module may have an optional class specification. If the class specification is not present, the first module with a matching name, regardless of its class, will be replaced in the library; otherwise only a module with a matching name and class will be replaced. The replace command will only replace a module whose name, or name and class (if both are specified), match a module already in the library.

Quit

This command quits the Library Manager, first saving the library file if it has changed. This command may be abbreviated to "q".

OMIT

This command directly exits the Library Manager without saving the library that was being edited. You may want to remember this one in case you hopelessly mess up a library file (although that shouldn't be cause for panic since the Library Manager always makes a backup file). Notice that there is no abbreviation for this command.

List {<module>{,<class>}}

The list command will print out information an the named modules. If no modules are specified, the list command will print out information on all of the modules in the library.

SList {<module>[,<class>]}

This is a short form of the List command. It prints out an abbreviated listing containing only the module name, class, and revision of each named module. If no modules are specified, this information will be printed for all modules in the library.

Help

The help command allows the user to obtain on-line help when using the Library Manager. It assumes there is a help text file available. The help command will print a menu and request a number from the user; it then prints the associated message and enters Interactive Mode.

LOad {<file>}

When anything is done involving a library which is currently not in memory, it is automatically loaded. The "load" command may be used to explicitly load a library without actually doing anything with it. Loaded libraries are not the same as the library you are editing; it is simply a library whose module information is in memory. When a module is from a library, for example, the module information for the entire library is loaded into memory so that the Library Manager can more quickly reference it. Before a file is loaded, the memory is checked to see if the file has already been loaded. A file is never loaded more than once. The "load" command may be abbreviated to "lo".

The reason a user may want to load a library explicitly is so the contents of a loaded library may be listed and examined using the load-list command as described below.

LList {<file>}

The LList command allows the user to list a loaded library. When used with a library name, the LList command will list the contents of the named library. When specified without any library name the LList command will list the names of all the currently loaded libraries. The "llist" command may be abbreviated to "ll".

SLList {<file>}
This command provides an abbreviated load-listing, including only the module name, class, and version. When this command is used without any library name specified, it will list the names of all currently loaded libraries. The "sllist" command may be abreviated to "sll".

Save {<file>)
The save command will force the Library Manager to save the library using the filename indicated by <file>. If no filename is explicitly specified, the library will be saved using the library name originally specified on the command line. As a safety measure, any time a file is saved the Library Manager will make a backup copy of any file which would have been overwritten by the save process. It will append a ".bak" extension to this backup file. The Library Manager will automatically save the library whenever the user exits using a "quit" command.

Comfile {<file>}
This command will direct the Library Manager to execute commands read from one or more specified files until it reads a "quit" or "omit" from the specified files or, alternatively until the end of the file is reached. An error message will be printed if no file is specified. The "comfile" command may be abbreviated by "c".

Echo {<any-string>}
This command simply echos the specifed strings to the terminal. This command can be useful in a command file to inform the user of its progress.

INTeractive
This command will explicitly place the Library Manager in Interactive Mode. Needless to say, it has no use when already in the Interactive mode, and very little use as a Direct Mode command (since the user can more readily enter Interactive Mode by simply not specifing any command whatever when calling the Library Manager). It is potentially useful in the Command File Mode, however, and can be included in a command file to force a return to the interactive Mode. The "interactive" command may be abbreviated as "int".

Find {<module>{,<class>]}
This command is used to "find" the module whose name and class is given.

There is a pointer in the Library Manager which points to what is known as the "current" module. When the Library Manager starts, the "current" module is the last-occurring module in the library being edited (assuming there are any modules in the library being edited). When an "Add" command is executed for example, the newly added modules are added following the "current" module. Almost every command has some effect on which particular module in the library is considered as being the "current" module after the commanded action has been completed. Following an add command, for instance, the

"current" module will become the last module that was added because of that add command. The list command also causes the current module to become the last module that is actually listed. In this manner, user command inputs continuously alter which specific module is actually considered the "current" module at any give time.

The find command can be used to explicitly define the current module to be any specific module in a library. Thus, if the user wishes to place a module in a specific place within the library, he can "find" the module which is to immediately precede the new module, and then "add" the new module. This will cause the new module to be placed immediately after the module that was "found" using the find command; this, of course, would also cause the "current" module to then become the newly added one.

The find command will attempt to move the "current" module pointer to the named module. It starts searching from the current module and continues until it reaches the bottom of the file, at which point it starts searching from the top of the file. It searches in this manner until it finds the named module, or until it reaches the original current module. If no module class is specified, the find command will stop at the first module it encounters that has the specified module name, regardless of its module class number; otherwise it will attempt to find a module which has both the name and class specified in the find command.

Print {<module>[,<class>]}
This command causes information to be printed for the named modules. If no modules are specified, it will print information on the "current" module.

SPrint {<module>[,<class>]}
This command works just like the Print command except it prints an abbreviated listing which includes only the name of the module, its class, and its revision.

Insert {<file>{,<module>f,[class>]}}
This command is similar to the "Add" command except, rather than placing the named modules after the "current" module, it will place them proceeding the current module in the library. When the Insert function finishes, the last module that was inserted then becomes the current module.

Stepping Through The Library
When editing a library using the Library Manager, a pointer exists which indicates the "current" module (as was described previously under the "find" command). This pointer is used as a starting point for searches when adding, exchanging, and deleting modules. It also points to the module which will be printed out by a "print" command when print is used without arguments. Most of the commands affect the value of this pointer, usually leaving it pointing to the last module that was referenced. There are several ways for the user to change the "current" module pointer. One is via the "find" command (see the Find command, above). For example, the following command

moves the painter to a module named "thing":

find thing

The user may also move the current module pointer around in a "relative" fashion by specifing a signed integer on the line. For example, the following will move the pointer backwards four (4) modules:

-4

By comparison an entry such as:

+2

will move the pointer forward two (2) modules.

It is also legal to specify one or more successive minus ("-")or plus ("+") signs to indicate the total number of modules to move backward or forward. For examole, a single minus or plus sign would move the pointer backward or forward one module, respectively. Two minus or two plus signs will move the pointer backward or forward two modules respectively (one for each symbol), and so on. It is also legal to move the pointer to a module located an absolute number of modules from the begining of the library; this is done by entering an unsigned number. For example, entering:

12

will move the pointer to the twelvth module in the library.

Any time one of these commands is executed, the Library Manager will print the name of the resultant current module. If one of these commands attempts to move the "current module pointer" above the top or below the bottom of the library, the Library Manager will print "TOP" or "BOTTOM" respectively.

CRstep

Executing this command toggles a flag which, when "on", causes a carriage return to act like a plus ("+") sign. This then allows a user to step down through the library, one module at a time, by simply hitting the carriage return. The CRstep command toggles this feature on (if previously off) or off (if previously on) with each execution. Therefore, if this feature has been previously selected to be "on", it can be selected to be "off" by simply re-entering the CRstep command once again.

QUIET

This command will prevent the Library Manager from printing out the name of the current module when the "current module" pointer moving commands are used. The "quiet" command may be abbreviated by "quie".

Additional Notes

If the user wishes to write out a module which is in a library, this

can be easily done by a command of the type:

```
libman newmod add oldlib,mod
```

For the filenames used in this example, this instructs the Library Manager to make a new library, called "newmcd", which contains a single module, called "mod", which was obtained from a library called "oldlib".

APPENDICES

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

APPENDIX A

LINKABLE FILE FORMAT

The following is the linkable file format which is expected by the Introl Linker and Loader.

There is no difference between a library file and a linkable object file as produced by the Assembler, other than the fact that a linkable object file contains only a single module whereas a library usually contains multiple modules. In the special case of a file which contains only a single module, it is permissible to have a text size specified as zero even though the text has a non-zero length. When a multi-byte value is specified, the most significant byte is assumed to appear first.

INTROL LINKABLE BINARY FILE FORMAT

HEADER

```
2 bytes Magic #
2 bytes Number of module descriptors in this file
1 byte Checksum of header
```

MODULE DESCRIPTOR (repeated for each module)

```
4 bytes Offset to module text in file
4 bytes Size of text (may be zero if
  single module in this file)
2 bytes Size of string area
1 byte   Module class
1 byte   Module revision
4 bytes  Relocatable segment @ax sizes

        |SF|SE|...|S7|S6|...|S0|

        Sn is a two bit max size  specifier:
                00   one byte max size
                01 - two byte max size
                10 - three byte max size
                11 - four byte max size

  4 bytes Relocatable segment  size descriptors

        |SF|SE|...|S7|S6|...|S0|

        Sn is a two bit descriptor size value:
                00 - no size
                01 - one byte size
                10 - two byte size
                11 - four byte size

 { 0..4 bytes segment 0 size }
 { 0..4 bytes segment 1 size }
                .
                .
                .
```

{ 0..4 bytes segment F size }

2 byte symbol count

For each symbol up to symbol count:

2 bytes Offset of identifier in string area
2 byte Descriptor value

|SZ|XXXXX|N|E|I|R|A|SEGM|

SZ is the descriptor of the symbol's value
00 - the value is zero
01 - the value follows in one byte
10 - the value follows in two bytes
11 - the value follows in four bytes

X is reserved
N set if the symbol is an entry point
A set if the symbol is absolute
E set if the symbol is exported
I set is the symbol is imported
(both E and I are set if the symbol is undefined segment imported)
SEGM is the segment the symbol resides in if non-absolute.

{ 0..4 byte symbol value }

The module descriptor string area starts here. The strings in the string area are null terminated ASCII character strings. The first string in the string area is the module name.

PROGRAM TEXT (follows all module descriptors in the file)

The basic text format is:

|CM|MODIFY| { 0 or more operand bytes }

CM is the two bit command.
MODIFY is 6 bits of command specific info.

code 00 - Special function

|00|FNCODE| {|function specific operands|}

FNCODE is a six bit special function code:

0 - end of text
1 - set byte size relocation
2 - set word size relocation

3 - set long size relocation

codes 4-15 are Loader commands

4 -reserved
5 -reserved
6 - "
7 - "
8 - "
9 - "
10 - "
11 - "
12 - "
13 - "
14 - "
15 - "

Multiple byte commands
The byte count is represented in the lower two bits as follows:

00 - the byte count follows in one byte
01 - the operand follows in one byte
10 - the operand follows in two bytes
11 - the operand follows in four bytes

16 - reserved
17 - skip with one byte byte count
18 - skip with two byte byte count
19 - skip with four byte byte count
20 - reserved
24 - reserved
28 - reserved

Segment set commands

32 - set segment 0
33 - set segment 1
34 - set segment 2
.
.
46 - set segment E
47 - set segment F
48 - reserved
49 - "
.
.
63 reserved

coce 01 - pass absolute text

|01|TCOUNT| |TCOUNT bytes of text|

TCOUNT - is the number of bytes to pass
(1-64). If TCOUNT == 0 then
byte count is 64.

code 10 - offset relocation command

|10|R|X|SEGM| |relocation size offset|

Relocation is done in the previously specified relocation size. The result is the proper relocated datum with the base of the given segment in this module added to the following offset. If the relative bit is set, the result is the proper relocated datum with the result being equal to the relocated value minus the value of the location counter follow- the relocated value.

R - set if the relocation is relative
X - is reserved
SEGM - is the segment # to relocate with

code 11 - symbol relocation command

|11|R|XX|S|OF |one or two byte symbol #| {|offset|}

Relocation is done in the previously specified relocation size. The result is the proper relocated datum with the result being equal to the value of the re- solved symbol plus the optional following offset. If the relative bit is set, the result is the proper relocated datum with the result being equal to the relocated value minus the value of the location counter following the relocated value.

R - set if the relocation is relative
XX - reserved
S - 0 if one byte symbol #, 1 if two byte sym. #
OF - size of the following offset

00 - zero offset
01 - byte offset
10 - word offset
11 - long offset

APPENDIX LF

FLD LOADER
OPTIONS AND RUNTIME ENVIRONMENT

The Introl Loader which generates Flex format output files is called the "fld" Loader.

The fld Loader is the "standard" Loader that is furnished with the part number FC6809 Introl-C Compiler and, as such, is the the Loader normally called by the FC6809's Linker when it finishes linking. The fld Loader is also optionally available for use with other versions of Introl-C (ie for Introl-C packages that do not themselves run under the Flex operating system) and, in such cases, is considered as being an "optional" Loader for these versions. (Refer to the "-d[<c>]" option discussed in the Linker section of this manual.)

The loader command line call for the fld Loader is of the form:

fld <filename> {<options>}

where <filename> is the module to be loaded and <options> are zero or more fld Loader option specifiers.

The fld Loader expects its input to be a relocatable module as produced by the Introl Linker, with any applicable "standard library", references having been being resolved using the FC6809 Standard Library. The fld Loader produces an output that is compatible with, and executable under, the Flex operating system. Executable files generated by the fld Loader are characterized by the filename extension ".CMD", which the fld Loader automatically appends to its output file.

Unless otherwise indicated, the following options for the fld Loader may be specified on either the linker command line (the typical case when the Loader is being automatically called by the Linker) or on the loader command line (when the Loader is being called independently by the user).

OPTIONS

-a=<sec>:<seg>{,<seg>}
: Assign segment to a section; where <sec> represents a Flex program segment which should be either "text", "data", or "bss", and <seg> is a segment number in the range 0 to 15. This option allows the user to override the default settings for placement of program segments.

-c=<file>
: Get additional parameters from a command file; where <file> is the command file filename. This option allows the user to specify an unlimited number of parameters by placing them, one to a line, in the named text file.

-l[s][=<file>]

Produce an output listing; where the "s" character is an optional entry, and <file> is an optional filename. This option forces the Loader to generate an output listing. If the optional s character is specified, the listing will contain symbol information. If the optional filename specification is included, the listing will be placed in the named file.

-o=<name>

Set output file name; where <name> is to be the name of the output file. If this option is omitted, the output file name will be that of the input name. If no filename extension is explicitly defined, the default extension ".CMD" will be assigned.

-W

Make an executable file no matter what! This option will cause the Loader to produce an executable output file even if there are still unresolved external references. It is not guaranteed as to what the result will be if the program actually attempts to access one of these unresolved items.

-y[{t|d|b}]=<origin>

Set origin; where the "t" or "d" or "b" character is optional, and <origin> is a hexadecimal number. This option may be used to set the origins of the text, initialized data, and uninitialized sections of the output file. If no t or d or b character is specified, or if the t character is specified, the text section will be placed at the location indicated by <origin>. If the d character is specified, the initialized data section will be placed at the location indicated by <origin>. If the b character is specified, the bss (uninitialized data) section will be placed at the location indicated by <origin>. if this option is not specifed, the text section will default be being placed at the zero origin, and will be immediately followed by the initialized data section, which will be immediately followed by the uninitialized data section.

-Z

Zap the input file. This option deletes the input file after the Loader has finished using it. When the Linker automatically calls the Loader, the Linker normally specifies this -z option as part of the call to cause the Loader to delete the file produced by the Linker (ie the intermediate ".RL" extension file) when it is no longer needed for loading purposes.

RUNTIME DATA MEMORY MAP

The runtime memory map shows the layout of the data space which a program has available during execution. The data appears in two areas, one of which is placed toward the low end of memory and another which is placed at the high end of memory (below the Flex operating system). The heap is placed in the low end of memory and grows upward by asking the operating system to enlarge its memory space. The stack is placed in the area at the high end of memory.

DATA MEMORY MAP

L.LF.4

INDEX