INMOS Linker Specification

Paul Sidnell, Andy Pepperdine and Andy Whitlow

SW-0012-9

INMOS Limited Confidential

APPROVED 6 February, 1991

1 Introduction

The Inmos linker **ilink** prepares object module files for the configurer or collector programs. It corresponds largely to the linker of traditional sequential systems and the output is to be loaded to only one transputer. The collector program (**icollect**) then prepares the program for a single transputer, or the configurer (**icconf**) is used to prepare several linked modules for execution on a network of one or more transputers.

The linker takes files in the TCOFF (see SW-0011) format and combines them to form a single output file in the same format. The files can be specified on the command line to **ilink** or in indirect files.

The previous format (LFF) defined in SW-0010 is accessible to the new **ilink** through a conversion program **icvlink** which will take LFF and produce the equivalent TCOFF. However, it is expected that the LFF will be phased out eventually. The ability of the linker to produce LFF output is retained for compatibility with the **iboot** and **iconf** programs of the previous toolset.

There is a facility for giving commands to **ilink** in an indirect file. This will allow extensible and extensive facilities to be made available for the user to modify such things as location values of otherwise undefined symbols, workspace requirements etc.

It must also be portable to as many systems as possible. This means that it is written in C, and is expected to run on SunOS, VMS, Ultrix, MS-DOS and Helios as well as root transputers attached to those systems.

The output of the linker must be for a specified target mode and processor or processor class. The linker will default to T414 Halt mode. If any of the input modules conflict with this, an error is produced. If a different mode or processor type is required it must be specified on the command line.

There is often a need to ensure that particular code occurs early in the linked code to allow it to be placed on the on chip ram. There is a predefined section name ”pri%text%base”, whose contents will (by default) be placed first in the code of the linked unit. If a compiler places a particular module in this section, it will be output before all modules that were not. This default may be overridden and replaced by a more complex ordering of multiple sections.

The previous linker produced a map file which contained information about each module and its position in the code. This information, which is needed by the debugger, will now be present within the linked unit in the form of a comment.

The new linker will have an option to produce a file that contains similar information but designed to be useful to the developer, containing information such as which library modules were pulled from the library and in which mode.

1.1 Limitations

ilink is not intended as a completely flexible and general purpose product. It does not perform functions that are more properly left to the linkers of more powerful operating systems. It only addresses the task of preparing a set of modules for stand-alone execution on a single node of a network.

The linker is designed to link modules whose target or mode attributes are encoded in such a way that they do not need to be interpreted, since a hierarchy is implicit in the encoding method. This is reasonable until an error must be generated concerning a disparity between the target attributes and the attributes of the input files.

The linker is aware of all present targets and modes and will display them in the error message. If an unrecognised attribute set is encountered in an error message, then the linker will decode only those attributes that it has knowledge of.

This means that should a new target class be introduced, the linker will continue to operate.

Note that files of the new format may be concatenated prior to linking to reduce the number of files used. This does not apply to libraries.

2 Parameters and Commands

The command line that invokes **`ilink`** also specifies certain parameters and files. The parameters all begin with the parameter indicator (which is **–** on Unix-like systems and **/** for VMS and MS-DOS). Other tokens are assumed to be the names of files. An invocation of the linker will produce a single linked file.

The named files may be linker input files, or indirect files. Linker input files are either TCOFF object files or library files; whereas indirect files are ASCII files. Indirect files can be introduced by the **`F`** parameter.

Indirect files contain lists of file names which are to be linked, or further indirect files to be read. Note that this lifts the present restriction on nested indirect files.

Indirect files also contain linker commands used to define symbol values, or rename symbols, etc. Such linker commands start with a hash (#) sign (like compiler directives). This technique has been chosen to allow for more extension of the facilities that the user may wish for later.

Comments can be included in indirect files by preceding them by two minus signs (**--**). They can occupy the whole line, or just the remainder of a line after a command or filename.

2.1 Parameters

Apart from the standard parameter (**`L`**), there are a number of parameters similar in function to those of the previous version of **`ilink`**.

`O filename:` this parameter defines the name of the file that is to receive the output from the linking process. If the file does not exist, then it is created. If it does exist, then it is overwritten. If the parameter is omitted, then the filename is taken from the first named file in the linker input but with a **`.lku`** extension. If this option is specified more than once, the latter is assumed.

`MO filename:` this parameter causes a link information file to be generated. This text file contains user information concerning the components and structure of the linked module.

`F filename:` this parameter specifies an indirect file. There are no restrictions, other than those of the host, as to what the filename might be. It is assumed to be a text file.

`I:` This standard option causes the linker to produce information concerning its version number and detailing what it is doing as it does it.

`L:` This standard option causes the tool to terminate immediately upon invocation. This option may only be used in conjunction with the **`I`**, **`XO`** and **`XM`** switches.

XO: This standard option causes the tool, upon termination, to terminate in such a way that the tool may not be re-run without being re-loaded. This is the default state and reflects the standard termination mechanisms of all supported environments.

XM: This standard option causes the tool, upon termination, to terminate in such a way that the tool need not be re-loaded. This option is transputer specific. The tool will loop within its harness and request from the iserver a fresh set of command line arguments.

U: this parameter indicates that any unresolved references at the end of linking are to be ignored. In this case, the resulting module cannot be re-linked. Warnings will still be generated for each unresolved reference.

ZI EnvironmentVariable: this parameter specifies a search path along which the linker will search for specified files. If this option is omitted, the variable **ISEARCH** is assumed. If this option is specified more than once, the latter is assumed.

H, S, X: these parameters specify the mode of the linked unit. **H** is the default and only one of these may be present. The linker will select appropriate modules from any libraries present. The parameters represent **HALT** mode, **STOP** mode and **UNIVERSAL** mode respectively.

Y: this parameter disables the interactive debugging capabilities of the linked unit. The linker will select appropriate modules from any libraries present. This option is not applicable to H series processors.

The following parameters select the target processor required. The linker will select appropriate modules from any libraries present. Note that the various transputer types presented here represent the current product range, not distinct instruction sets.

T2, T212:	for T212 compatibility.
T222:	for T222 compatibility.
M212:	for M212 compatibility.
T3, T225:	for T225 compatibility.
T400:	for T400 compatibility.
T4, T414:	for T414 compatibility.
T5, T425:	for T425 compatibility.
T8, T800:	for T800 compatibility.
T801:	for T801 compatibility.
T9, T805:	for T805 compatibility.
TA:	for TA compatibility.
TB:	for TB compatibility.
TH, TH1:	for H1 compatibility.

KB memory: If the linker is running on a machine with limited memory available, a limit may be applied to the amount of the linked program image that is kept in internal store. If the image is larger than this limit, the remainder is stored in a temporary file. The minimum amount is 1k.

T: this parameter causes TCOFF output to be generated, this is the default output format.

LB: this option causes LFF output to be generated for the **iboot** tool.

LC: this option causes LFF output to be generated for the **iconf** tool.

ME entryname: this option specifies the name of the main entry point of the program and is equivalent to the **#MAINENTRY** command (see later).

EX: this parameter allows the extraction of modules unlinked. The linker functions as normal except that the output will not be a single linked unit. It will in fact be the concatenation of the component modules that would have made up the linked unit, with each being unaltered. This is used for creating convienient sub units for further linking, or for extraction from libraries. The **U** option, **#REFERENCE** and #DEFINE commands are particularly useful for controlling the content of the output.

- **U** will allow the linker to continue even when unresolved references remain. These outstanding references remain unresolved in the output and must be resolved in some subsequent link.
- **#REFERENCE** causes the linker to pull in the module exporting that name, and hence everything subsequently required by it.
- **#DEFINE** resolves a reference causing the linker not to pull in any module because of that symbol. Remember that there may be other external symbols in that module which may also require this. In the case of occam specific references this is not possible, see below. The actual value used in the **#DEFINE** command is not significant, since no patching is done.

A main entry point need not be specified with this option and, will have no effect.

In the case of occam, for safety reasons the linker will not allow the separation of a module, and those modules which it specifically references, they may be considered permanently fused.

2.2 Commands

#ALIAS base_name [alias_name]...

The alias command can occur in any indirect file. It defines a list of names which act as aliases for a **base_name**. A symbol is created for each alias, and is linked to the base name, allowing the alias name to take on the value of the base name when it is defined.

This command can be used to resolve the references which would otherwise be unresolved or override a particular resolution. for example, if a module contains a call to routine "call_proc" then some other function may be aliased to "call_proc", causing the call to be made to a different procedure, eg **#ALIAS new_proc call_proc**. The original reference is considered resolved and will not cause any modules to be loaded in from a library. If the alias name is exported by any of the users modules then it is an error as the symbol will be multiply defined.

#DEFINE name int_value

This defines **name** as having the value of **int_value**, which must be either an optionally signed decimal integer, or unsigned hexadecimal (if preceded by a # sign) integer. The value must fit into the word length of the target machine.

This can be used by compilers to define such things as stacksizes at link time. A default could be provided in the language's library by a suitable definition of the symbol at the end, after all other symbols have been defined.

#REFERENCE name

This command creates a reference to the given name. It does not allocate any storage or change any text. The purpose is to allow the inclusion of library initialisation routines which might not otherwise be included because there are no references to them.

#MAINENTRY name

This command specifies the symbol which is the main entry point for the program. It is an error if that symbol is not subsequently found in the input.

#SECTION name

This command causes the linker to place the named section first in the code of the linked unit. This overrides the default section "pri%text%base". If more than one of these commands are supplied, then the named sections are output before any unnamed sections, and in the same order as the commands.

The order of modules within a section is the order in which they were linked.

3 Output model

The linker combines a number of modules into a single one. It will then contain one section, containing both code and data, and another virtual section into which are mapped all the Fortran COMMON blocks and C static data items. The individual sections in the TCOFF will be kept separate until the final merging. This will ensure that any language that relies on building a single section from directives in several different modules (eg. module tables) can still do so. However, such a module cannot then be further linked with others which wish to add to these sections.

Three output formats are possible. TCOFF output for the new tools (the default), and two forms of LFF output. LFF output may be generated for a simple configuration, ie. there is a single main entry point in the output - suitable for **iboot**. The second form of LFF is for when the old configurer is to be used, and all available entry points will be present in the output file.

3.1 Output File

The contents of Fig.1 is the result of using the **ilist** tool on a TCOFF linked unit file. It is suggested that this description of the linked unit file be read in conjunction with document **SW-0011**, the Transputer Common Object File Format description.

The linked unit will begin with a **LINKED_UNIT** tag to distinguish it from any other form of TCOFF file.

Compilers may use virtual sections to model the run time memory usage of a program in order to calculate patch values. At present, such sections are used to model Fortran common blocks, C static data usage and module tables. The size of a virtual section represents the number of words that are known to be required at run time.

Tools that process linked units will need the total memory usage of a program. To allow this, the linker will produce a single virtual section, whose size is the sum of all virtual sections present in the linked modules.

The sections that contributed to this summing process will be converted to local symbols with the same name. Their values will reflect their size in words at the end of linkage. Note that this is for information purposes only.

The program code is loaded into an executable text section with a single **LOAD_TEXT** directive.

There is a symbol representing the main entry point of the program. Its value is the offset in bytes from the start of the program code to the entry point itself. This symbol is identified by the **DEFINE_MAIN**

```
00000000 LINKED_UNIT
00000002 START_MODULE T CORE FMUL BIT32 MS<=28 H lang: LINKED ""
00000010 VERSION tool: ilink origin: <various>
00000022 COMMENT PRINT "LINKED FILE"
00000032 SYMBOL LOC "module%table%base" id: 0
00000047 DEFINE_SYMBOL id: 0 67
0000004C SYMBOL LOC "static%base" id: 1
0000005B DEFINE_SYMBOL id: 1 2126
00000062 SECTION VIR "TOTAL%STATIC" id: 2
00000073 SET_LOAD_POINT id: 2
00000076 ADJUST_POINT 2193
0000007C SECTION REA EXE EXP "linked%text" id: 3
0000008C SET_LOAD_POINT id: 3
0000008F LOAD_TEXT bytes: 66928
000105FE SYMBOL EXP ORI "file.lku:972586" id: 4
0001060A SYMBOL EXP "MAIN.ENTRY" id: 5
00010618 DEFINE_SYMBOL id: 5 39286
0001061F DESCRIPTOR id: 5 lang: OCCAM
ws: 491 vs: 0
PROC MAIN.ENTRY(CHAN OF SP FromFiler,
                CHAN OF SP ToFiler,
                []INT FreeMemory,
                []INT StackMemory)
  SEQ
    FromFiler?
    ToFiler!
:
000106A7 DEFINE_MAIN id: 5
000106AA COMMENT PRINT "LINKER TCOFF
SC  ilink.tco     (0000002) 000000 019311 : -
SC  expr.tco      (0000002) 019312 023503 : -
SC  module.tco    (0000002) 023504 032999 : -
      ... etc.
LIB libc.lib      (0997110) 057204 060523 : sub_vfprintf%c
LIB libc.lib      (0922358) 060524 060639 : unlink%c
LIB libc.lib      (0911046) 060640 060723 : IsRemove%c
LIB libc.lib      (1007529) 060724 060891 : writebuf%c
LIB libc.lib      (0924094) 060892 061227 : write%c
LIB libc.lib      (0904581) 061228 061299 : IsFflush%c
LIB libc.lib      (0910493) 061300 061399 : IsFwrite%c
"
00011EA2 END_MODULE
```

Figure 1: Linked Unit Contents

directive.

If the main entry point has a descriptor, this will also be present.

A single symbol with the **ORIGIN** attribute will be present in the file. This will have a 'unique' name and may be used by other tools for verification. The format of the symbol name is a sequence of characters followed by a colon followed by a hexadecimal number, not exceeding 32 bits.

A printable comment containing mapping information will be present. This comment is in a similar format to the map file of the previous toolset. Each line has a maximum length of 256 characters, including the terminating newline (character 10).

This map comment will only exist until debug sections are implemented.

4 Error Messages

The following is a list of error messages and a brief description of their meaning.

4.1 General

Serious-ilink-<file>-mode: <mode> linker mode: <mode>
The linker has been given a module to link which has been compiled with attributes incompatible with the options (or lack thereof) on the linker command line.

Serious-ilink-<file>-could not open for input
Serious-ilink-<file>-could not open for output
The named file could not be found/opened for reading/writing.

Serious-ilink-nothing to link
Various options have been given to the linker but no modules or libraries.

Serious-ilink-cannot create output without main entry point
No main entry point has been specified.

Serious-ilink-multiple main entry points specified
The main entry point has been specified on the command line or in an indirect file more than once.

Warning-ilink-<size> bytes too large for 16 bit target
The code part of the linked unit has exceeded the address space of the T212 derived processor family.

Serious-ilink-<file>-bad format: not linkable file or library
The linker expects that all file names presented without a preceeding switch (on the command line) or directive (in an indirect file) are either libraries or modules.

Error-ilink-<file>-symbol <symbol> multiply defined
The symbol, introduced in the specified file, has been introduced previously, causing a conflict. The same module may have been supplied to the linker more than once or there may be two or more modules with the same entry point or data item defined.

Warning-ilink-<file>-symbol <symbol> not found
Error-ilink-<file>-symbol <symbol> not found
The specified symbol was not found in any of the supplied modules or libraries. This is only a warning

if the **U** switch is used.

Serious-ilink-<file>-message: <message>
Warning-ilink-<file>-message: <message>
Message-ilink-<file>-<message>
Tools that create modules to be linked with ilink may embed "messages" within them. Three levels of severity exist. The documentation of the appropriate tool should be consulted for more information.

Error-ilink-<file>-usage of <symbol> out of step with <file>
Languages such as OCCAM have a **#USE <filename>** directive which causes the compiler to scan the **filename** for details concerning certain program resources. It is therefore essential that this file be unchanged at link time. This diagnostic indicates that this is not the case. There are several possible causes:

- File2 has been recompiled after file1, in which case file1 requires recompiling.
- The file that occurred in the **#USE** directive has been replaced by a different version of the file at link time.
- The file that occurred in the **#USE** directive has not been supplied to the linker, but the linker has located a different version of a required entrypoint elsewhere.

Error-ilink-<file>-name clash with <symbol> from <file>
In languages such as OCCAM entrypoints may be scoped, ie. extra information is associated with each symbol to indicate which version of that entry point it is. This allows programs to be safely linked even though there are several different versions of the same entrypoint occurring at different lexical levels within the program.

This error indicates that a language without such scoping has been mixed with a scoped language such as OCCAM and a name conflict has occurred between a scoped and non scoped symbol.

Serious-ilink-invalid or missing descriptor for main entry point <symbol>
The specified main entry point to the program does not have a valid OCCAM descriptor. This occurs if the wrong symbol name has been used, for example a data symbol in C.

Serious-ilink-<file>-bad format: multiple main entry points encountered
A symbol may be defined to be the main entry point of a program by a compiler. Only one such symbol must exist within a single link.

Serious-ilink-could not open temporary file
When using the **kb** option the linker may (if the image being linked exceeds the size specified) need to create a temporary file. The tool was unable to do this. Note that on some systems this temporary file will be in the current working directory, in which case the user would have to have write access.

4.2 Command line

Serious-ilink-command line: image limit multiply specified
Serious-ilink-command line: multiple debug modes
Serious-ilink-command line: multiple error modes
Serious-ilink-command line: multiple module files specified
Serious-ilink-command line: multiple output files specified
Serious-ilink-command line: multiple target type
Serious-ilink-command line: only one output format allowed
Serious-ilink-command line: 'load and terminate' option set, some

arguments invalid
Serious-ilink-command line: y mode inapplicable to H targets
The above errors correspond to illegal combinations of command line arguments.

Serious-ilink-command line: <token>
An illegal token has been encountered on the command line.

Serious-ilink-command line: bad format number
A numerical parameter of the wrong format has been found.

Serious-ilink-command line: 1k minimum for paged memory option
When using the **KB** option, the amount of memory used to hold the image of the program being linked is specified. There is a minimum size of 1k.

4.3 Indirect files

Serious-ilink-<file>-<line>-could not open for reading
The file name specified in an **INCLUDE** directive could not opened.

Serious-ilink-<file>-nothing of importance in file
The file name specified in an **INCLUDE** directive was empty or contained nothing but white space or comments. Perhaps this should be a warning - I leave this as a legacy to future generations.

Serious-ilink-<file>-<line>-unrecognised directive <directive>
An unrecognised directive has been found in an indirect file.

Serious-ilink-<file>-<line>-only one file name per line
More than one file name has been placed on a single line within an indirect file.

Serious-ilink-<file>-<line>-bad format: only single parameter for <directive>

The directive has been given too many parameters.

Serious-ilink-<file>-<line>-bad format: non ASCII character in indirect file

The indirect file contains some non printable text. A common mistake is to specify a library or module with the **F** command line argument or the **INCLUDE** directive.

Serious-ilink-<file>-<line>-bad format: file name missing after <directive>
A directive (such as **INCLUDE**) has no file name as an argument.

Serious-ilink-<file>-<line>-bad format: excessively long line in indirect file
A line is too long. The length is implementation dependant, but on all currently supported hosts it is long enough so as only to be exceeded in error.

Serious-ilink-<file>-<line>-bad format: <directive> invalid number
A numeric parameter supplied to a directive does not correspond to the appropriate format.

Serious-ilink-<file>-<line>-<directive> not enough arguments
Serious-ilink-<file>-<line>-<directive> too many arguments
The wrong number of arguments have been supplied to a directive.

4.4 LFF format

Serious-ilink-unknown error modes not supported in the LFF format
Serious-ilink-unknown processors not supported in the LFF format
When generating LFF format files, certain constructs will have no representation. For example processor types that have come into existance since the LFF format was defined.

Warning-ilink-<file>-<symbol>, implementation of channel arrays has changed
LFF files are often generated so that the LFF configurer may be used, but it should be noted that channel arrays should not be used as parameters to configured procedures since they are implemenented differently in the new OCCAM compiler and the old configurer.

4.5 System failures etc.

Fatal-ilink-no more files available
The linker has attempted to exceed the number of files available on the host operating system. **INCLUDE** directives are stacked and no matter how deeply they are nested, only one will ever be open. Module files are only open while they are read. All library files are kept open indefinitely, unless a subsequent file open operation fails, in which case an already open library file is closed and the operation retried. The linked unit and module file are written out sequentially. This accounts for a single file only. If the **KB** option is used, a page file is kept open for the entire link. Hence the linker should be capable of continuing in all circumstances with a minimum of two files available. Unless there is something wrong of course.

```
Warning-ilink-<file>-bad format:  unknown record type <record> ignored
Warning-ilink-environment variable <variable> nonexistant
Warning-ilink-<file>-text not word aligned - has been extended
Serious-ilink-<file>-bad format:  illegal usage for symbol <symbol>
Serious-ilink-<file>-bad format:  conflicting usage for symbol <symbol>
Serious-ilink-<file>-bad format:  unexpected end of file
Serious-ilink-<file>-bad format:  symbol <symbol> undefined
Serious-ilink-<file>-bad format:  section out of range in expression
Serious-ilink-<file>-bad format:  symbol out of range in expression
Serious-ilink-<file>-bad format:  unknown operator in expression
Serious-ilink-<file>-bad format:  unrecognised operator in expression
Serious-ilink-<file>-bad format:  library inconsistant with index
Serious-ilink-<file>-bad format:  library inconsistant with index
Serious-ilink-<file>-bad format:  unknown record in library index
Serious-ilink-<file>-bad format:  BYTE_PATCH illegal in virtual section
Serious-ilink-<file>-bad format:  LOAD_EXPR in virtual section
Serious-ilink-<file>-bad format:  LOAD_PREFIX illegal in virtual section
Serious-ilink-<file>-bad format:  LOAD_TEXT in virtual section
Serious-ilink-<file>-bad format:  LOAD_ZEROS illegal in virtual section
Serious-ilink-<file>-bad format:  WORD_PATCH illegal in virtual section
Serious-ilink-<file>-bad format:  invalid load point before LOAD_EXPR
Serious-ilink-<file>-bad format:  invalid load point before LOAD_PREFIX
Serious-ilink-<file>-bad format:  invalid load point before LOAD_TEXT
Serious-ilink-<file>-bad format:  invalid load point before LOAD_ZEROS
Serious-ilink-<file>-bad format:  invalid load point brfore DEFINE_SYMBOL
Serious-ilink-<file>-bad format:  invalid load point
Serious-ilink-<file>-bad format:  local symbol in KILL_ID
Serious-ilink-<file>-bad format:  multiple origins exported
Serious-ilink-<file>-bad format:  symbol out of range in DEFINE_MAIN
Serious-ilink-<file>-bad format:  symbol out of range in DEFINE_SYMBOL
```

```
Serious-ilink-<file>-bad format:  symbol out of range in DESCRIPTOR
Serious-ilink-<file>-bad format:  symbol out of range in KILL_ID
Serious-ilink-<file>-bad format:  symbol out of range in SET_LOAD_POINT
Serious-ilink-<file>-bad format:  symbol out of range in SPECIFIC_SYMBOL
Serious-ilink-<file>-bad format:  unknown message type
Fatal-ilink-<file>-ALIGN not yet supported
Fatal-ilink-<file>-additive patches not yet supported
Fatal-ilink-<file>-replicators not yet supported
Fatal-ilink-<file>-minimum prefixes not yet supported
Fatal-ilink-<file>-bad patch at address <address>
Fatal-ilink-<file>-bad patch address
Fatal-ilink-<file>-symbol <symbol> has bad value
Fatal-ilink-<file>-patch too long at <address>
Fatal-ilink-<file>-patch too long
```

The above errors are not expected during normal operation. They should only appear if an invalid module is presented to the linker, or a non-user option has been used incorrectly.

```
Fatal-ilink-internal error <number>:  please report
Fatal-ilink-internal command file error <number>:  please report
```

These messages should never be displayed during normal operation. Each number is unique and corresponds to an specific area of code. No description is given here since the cause of the problem is unknown and the code will need inspection.

5 Present Implementation Restrictions

Only those directives currently needed have been implemented. This excludes the following directives:

5.1 nested modules

Not implemented.

5.2 local_symbols

Not implemented.

5.3 load_prefix

Only fixed length prefixes are implemented.

5.4 align

At present it is the compiler's responsibility to ensure that its code is aligned, and that each modules code terminates on a word boundary. While this situation exists, the linker will check that every text section is aligned at the end of each module. If not, it will warn and pad with 0x20 up to the next word boundary.

5.5 comment

The copy attribute is ignored and assumed not to be set.

5.6 rep_start

Not implemented.

5.7 rep_end

Not implemented.

5.8 debug sections

Not implemented.