Intel® Linker

User's Manual

September 2003

Revision 2.0

Order Number: 278467-005

Contents

Figures

Tables

Revision History

Date	Revision	Description
May 2001	1.0	Release for Intel® Linker version 0.01 and higher.
August 2002	1.1	Release for Intel® Linker version 0.02 and higher.
March 2003	1.2	Release for the Intel® Linker version 1.1 and higher.
July 2003	1.3	Release for the Intel® Linker version 1.1 and higher.
September 2003	2.0	Release for the Intel® Linker version 1.2 and higher.

This page intentionally left blank.

Introduction 1

The Intel® Linker is a tool combining linker and builder. It is designed to build C/C++ embedded applications for the Intel XScale® Microarchitecture. The Intel® Linker processes relocatable input files and creates a single executable file. The object format of these files is the ELF/DWARF object format. The Intel® Linker works in *final link mode*, where the logical section fragments are bound to physical segments at absolute addresses. All relocation is completed, and a single absolute symbol table is written out. The Intel® Linker supplies the following features:

- ELF/DWARF object format support
- Public-external resolution
- Relocation processing
- Assignment of addresses
- Construction and destruction of static C++ objects
- Command line options to control the link process
- Creation of debug information
- Creation of a listing file
- Build file to apply specific settings to the output file

The Intel® Linker is part of the Intel® C++ Software Development Tool Suite consisting of the following tools:

- Intel® C++ Compiler
- Intel® Assembler
- Intel® Linker
- Intel® Library Manager
- Intel® Object Converters

1.1 About this Manual

This is a reference manual for the Intel® Linker on Windows* host platforms. The manual contains the following chapters:

- Chapter 1, “Introduction” is this chapter. It also includes related documentation, overview of requirements and the manual’s conventions.
- Chapter 2, “General Usage” describes the general usage of the Intel® Linker. This includes command line syntax, input and output files, overview of options, linking process, linking libraries, and the allocation of segments and sections.
- Chapter 3, “Options” provides a summary of options, together with their default settings. The remaining sections describe the options in detail, which are sorted alphabetically.
- Chapter 4, “The Build File” describes directives that are available to create a build file. Such a build file can be used to specify options in addition to those on the linker command line. These options are used within the linking process. The build file can contain, for example, specific settings for alignment of segments and sections, or the layout of your linker output.
- Chapter 5, “Listing File” describes the listing file that may be created during the link process. The chapter explains the creation of a listing file as well as its contents.
- Chapter 6, “Diagnostic Messages” describes the format of diagnostic messages.
- Chapter , “Appendix – Reserved Words” contains the list of reserved words. These words must be preceded by an escaped character if they are used inside build files.

1.2 Related Documentation

This section provides an overview of documentation which supplements this manual. These are:

- The *Intel® C++ Compiler User's Manual. For Intel XScale® Microarchitecture*, order number 278496
- The *Intel® Assembler Reference Manual*, order number 278586
- The *Intel® Library Manager User's Manual*, order number 278468

Furthermore, Release Notes may be provided. Release Notes contain features or changes of the product that are not documented in the corresponding manual. Release Notes are included in the documentation set of your installation.

There are a few other documents which provide related information. These are:

- TIS Committee, 1995, *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, Version 1.2*
- TIS Committee, 1995, *DWARF Debugging Information Format Specification, Version 2.0*
- ARM* Limited, 2001, *ARM ELF*
- ARM* Limited, 2000, *The ARM-THUMB Procedure Call Standard*

1.3 Requirements

To use the Intel® Linker, the following environment is required:

- Microsoft* Windows* 2000 Professional or Windows* XP Professional as host platform

1.4 Conventions

This manual uses several notational and typographical conventions to visually differentiate text, these are explained in the table below.

Convention	Description
Italics	Italics identify variables and introduce new terminology. Titles of manuals are in italic font. Italics are used for emphasis.
Bold	Used to specify something important.
`Plain Courier`	Used to specify written code.
`Italic Courier`	Used to specify parameters.
GUI-elements	Elements of the graphical user interface are written in this style.
<>	Filenames and function names to be defined are delimited by <>.
|	Used to specify an alternative between several items.
,...	Used to specify an item which may be repeated.
[ ]	Used to specify optional items.
{ }	Used to specify an optional item which may be repeated.
"[" "]" "{" "}" "|"	Writing a metacharacter in quotation marks negates the syntactical meaning stated above; the character is taken as a literal. For example, the line "[" X "]" [Y] denotes the letter X enclosed in brackets, optionally followed by the letter Y.

This page intentionally left blank.

General Usage 2

This chapter provides general information on the usage of the Intel® Linker. The chapter contains the following topics:

- Section 2.1, “Command Line Syntax” on page 14 describes how the Intel® Linker is started on a command line.
- Section 2.2, “Input and Output Files” on page 16 describes the input and output files the Intel® Linker requires to work with.
- Section 2.3, “Overview of Options” on page 17 provides an overview of all available command line options, including a brief description of each.
- Section 2.4, “The Process of Linking” on page 20 explains the different passes the linker processes to create a linked output file.
- Section 2.5, “Linking Libraries” on page 21 explains how libraries can be linked.
- Section 2.6, “Allocation of Segments and Sections” on page 22 describes the default allocation of segments and sections. An example shows how the default allocation may be changed.

2.1 Command Line Syntax

The Intel® Linker is started using the following call:

```
ldxsc [options] {objectfiles}
```

where:

`ldxsc`	`ldxsc` is the executable file of the Intel® Linker.
`options`	Specifies command line options. If several options are used, they must be separated by spaces. For an overview of command line options see Section 2.3, "Overview of Options" on page 17. For a detailed descriptions of command line options seeChapter 3, "Options".
`objectfiles`	Specifies ELF/DWARF relocatable object files and/or library files that are linked. You must specify one object file on the command line at least. The object file can also include a path name, e.g. ./test/myobject.o.

Example 1. Simple Way of Linking

```
ldxsc a.o b.o
```

The command line above links the two objects a.o and b.o to the output file a.x.

Note: By default, the name of the output file is the name of the first module to be linked.

2.1.1 Order of Arguments

The order of arguments, options, and files is of no consequence; it is even possible to specify object files before the options or between two options.

2.1.2 Option Arguments

Option arguments *`options`* must each begin with a minus sign '`-`' or a slash '/'. They modify the default actions of the Intel® Linker. The available options are described in Section 2.3, "Overview of Options" on page 17 and Chapter 3, "Options".

It is also possible to use option files which contain more option arguments. These option files must start with an "@" character to be treated as files which contain options. Option files and command line options may be used together.

The file format of an option file is rather free. The options may be written as on the command line, but it is possible to set a "line feed" at any place where a space is permitted in the command line. It is recommended to write each option together with its argument(s) onto a separate line, together with comments describing their actions.

Example 2. Option File

The following command lines have the same effect:

```
ldxsc -buildfile=link.bld -verbose -cplus a.o b.o
```

and

```
ldxsc -buildfile=link.bld @linker.opt a.o b.o
```

where the last command line uses the option file linker.opt, which has the following contents:

```
-verbose            ; print messages about linker passes
-cplus                ; output C++ definitions
```

2.2 Input and Output Files

The Intel® Linker accepts the following input files:

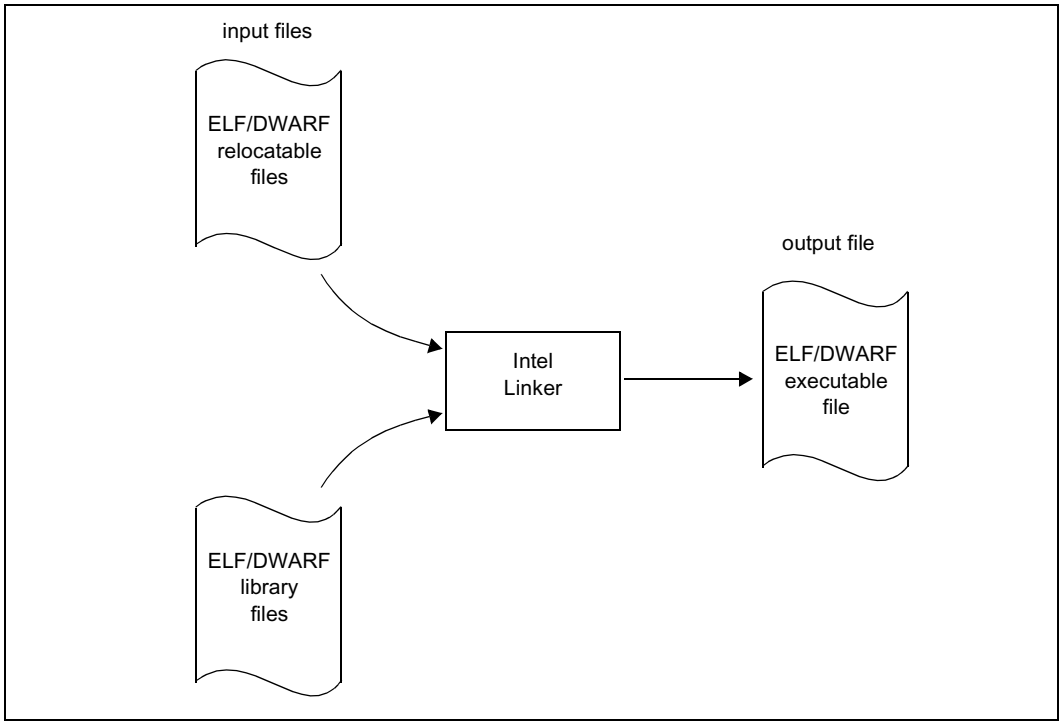
- ELF/DWARF relocatable files. The default extension of these files is .o.
- ELF/DWARF library files. The default extension of these files is .a or .lib.

The Intel® Linker produces an *ELF/DWARF executable file*. The default extension of this file is .x.

An *ELF/DWARF relocatable file* is an object file which contains code and data for linking. The object format of this file is the ELF (Executable and Linking Format). If the object file contains debug information, one or more sections are added to the ELF object file containing the DWARF debugging information format.

An *ELF/DWARF library file* may contain modules with common characteristics (i.e. modules that perform similar functions or are required for a particular system). For example, a library might contain modules that perform mathematical functions and another might contain modules that perform I/O routines. The object format of these modules is the ELF.

Figure 1. Input Files and Output File



2.3 Overview of Options

This section provides a table giving an overview of available command line options in alphabetical order. For an overview in usage order and a detailed description of each of the options refer to Chapter 3, “Options”.

Table 1. Option Overview

Option	Abbreviation	Description
@filename		Specifies an option file. See Section 2.1.2, “Option Arguments” on page 14
-base *arg*	-b *arg*	Sets the base address to the absolute value *arg*. *arg* can be a hexadecimal, decimal, or octal absolute value. Default base address is 0x8000.
-buildfile	-bf	Processes the default build file. The default build file name is the base name of the output file with the extension .bld.
-buildfile=*name*	-bf=*name*	Processes the specified build file *name*.
-buildmacro *macroname*[=*text*]	-bm *macroname*[=*text*]	Enables usage of macros inside the build file.
-cdtorseg *arg*	-cds *arg*	Changes the default allocation of constructor and destructor tables. This option is only valid if -cplus is set.
-cplus	-cp	Creates tables for constructors and destructors of static C++ objects.
-commalign=*number*	-ca=*number*	Selects the global alignment for common data.
-ctorpattern *arg*	-ctp *arg*	Specifies the pattern for constructor callers. This option is only valid if -cplus is set.
-ctortab *arg*	-ctt *arg*	Specifies the symbol name of the constructor table. This option is only valid if the option -cplus is set.
-debug	-db	Links debug information to the output file.
-dtorpattern *arg*	-dtp *arg*	Specifies the pattern for destructor callers. This option is only valid if -cplus is set.
-dtortab *arg*	-dtt *arg*	Specifies the symbol name of the destructor table. This option is only valid if the option -cplus is set.
-dynamic	-dyn	Causes any shared object file to be generated in the Linux* format.
-dynamic=*arg*	-dyn=*arg*	Causes any shared object file to be generated in the format specified by *arg*, where *arg* can be Linux or ARM.
-entry *arg*	-e *arg*	Defines the entry point. *arg* can be a symbol name or an address.

Table 1. Option Overview

Option	Abbreviation	Description
-export *arg*{,*arg*}	-ex [*arg*],...	Specifies global symbols that are to be written to the final symbol tables. These symbols may be exported to other modules.
-forceblx	-fblx	Changes BL instructions to BLX to avoid generation of veneer code.
-help	-?	Prints a help list on the screen.
-import *arg*{,*arg*}	-im *arg*{,*arg*}	Specifies global symbols that are to be written to the final symbol tables. These are symbols which require to be imported from other modules.
-info=*arg*{,*arg*}	-i=*arg*{,*arg*}	Specifies parameters for the listing file. Available values for *arg* are: errors, segments, sections, modules, memory, symbols, map, and buildfile.
-init *arg*{,*arg*}	-in *arg*{,*arg*}	Initializes the specified segments.
-keep		
-listing	-l	Creates a listing file.
-listing=*name*	-l=*name*	Creates a listing file with name *name*.
-mapcomdat	-mcd	Maps global common data to the first input module that is referring to it.
-noinfo	-ni	Suppresses the output of information messages.
-normempty	-nre	Suppresses the removal of empty sections.
-nosymbols	-ns	Suppresses the generation of the symbol table.
-nowarnings	-nw	Suppresses warning messages.
-output *name*	-o *name*	Specifies the output filename.
-remove[=*arg*{,*arg*}]	-rm[=*arg*{,arg}]	Removes all non-referenced input sections, of types specified, or all types if none specified.
-ropi		Generates relocation information for position-independent read-only segments.
-rwpi		Generates relocation information for position-independent read-write segments.
-shared	-sh	Generates a shared object file.
-strict	-sc	Forces strict consistency checking of the linked modules.

Table 1. Option Overview

Option	Abbreviation	Description
`-unref`	`-ur`	Displays warning messages for unreferenced symbols.
`-verbose`	`-v`	Prints additional information on the link process on the screen. This option is the same as `-verbose=0`.
`-verbose=`*num*	`-v=`*num*	Prints additional information on the link process on the screen. Additionally, you can specify by *num* which information is to be displayed. Possible values for *num* are 0, 1, 2, 3, and 4.

2.4 The Process of Linking

The Intel® Linker performs its tasks in 4 passes. To make them visible, the `-verbose` command line option needs to be specified. These passes are as follows:

```
LDXSC: PASS 0 : scan input ...
LDXSC: PASS 1 : layout ...
LDXSC: PASS 2 : load and relocate ...
LDXSC: PASS 3 : write output ...
```

The first pass (`PASS 0`) scans the linker commands, which are checked syntactically and semantically. Input files are opened and checked for correct formatting. If the processed file is an object file, the header information of the section size is read in and the symbol table is added to the Intel® Linker's internal symbol table. If the file is an object library, it is searched for symbols, satisfying currently undefined references. Such modules are added in the manner mentioned above. The Intel® Linker reclaims undefined symbols and exits at the end of this pass.

The second pass (`PASS 1`) performs the following actions according to the linker commands which are used in the command file:

- The base addresses and sizes of included modules are calculated.
- In the case of a final link the physical layout of the executable file is calculated.
- Quota and overlap checks are calculated.

The third pass (`PASS 2`) collects the section fragments, relocates references, and constructs the symbol table.

The last pass (`PASS 3`) writes all that has been carried out in `PASS 2` to the output file in the corresponding format.

Further Information:

Section 3.36, "-verbose" on page 70
Section 3.36, "-verbose" on page 70

2.5 Linking Libraries

The library is searched for an index section at the first position. This section is an index of external symbols defined within the relocatable objects which are contained in the library.

The Intel® Linker scans this index, looking for any undefined symbols in its symbol table. If such a symbol is found, the object module defining it is extracted from the library and linked into the program. This is the same process as when any other object module is linked. When the module has been linked successfully, the search continues until all symbols currently marked as undefined references have been resolved.

Note that when a module is linked, several undefined references may be satisfied at once and additional undefined references may be added. This implies that library functions may refer to symbols defined within other library functions. When the module containing the first function is added, the undefined symbols of this module are appended as undefined references to the internal symbol table of the linker. This may cause the linker to add the modules containing these symbols in the same manner.

The search process terminates when the internal symbol table of the linker contains no more undefined references or when the search through the library index is finished and no additional objects have been included.

Note that the Intel® Linker cannot deal with libraries containing object modules with global symbols but no library index. This also applies to libraries containing only modules without any global symbols.

2.6 Allocation of Segments and Sections

The linked file consists of a set of segments within which are sections.

For embedded developing it is sometimes necessary to allocate a section within a specific segment, or to allocate the segments at a specific piece of memory. Therefore, you need to know which sections and segments are supplied by the compiler and where these are allocated by default. The default allocation can globally be changed inside the PARAMETER block in the build file or individually inside the LAYOUT block.

Further Information:
Section 4.2.1, “PARAMETER Block” on page 79
Section 4.2.2, “LAYOUT Block” on page 84

2.6.1 Predefined Sections

The Intel® C++ Compiler provides a set of sections with different attributes, as shown in the following table. The third column “Segment” shows the name of the segment the section belongs to.

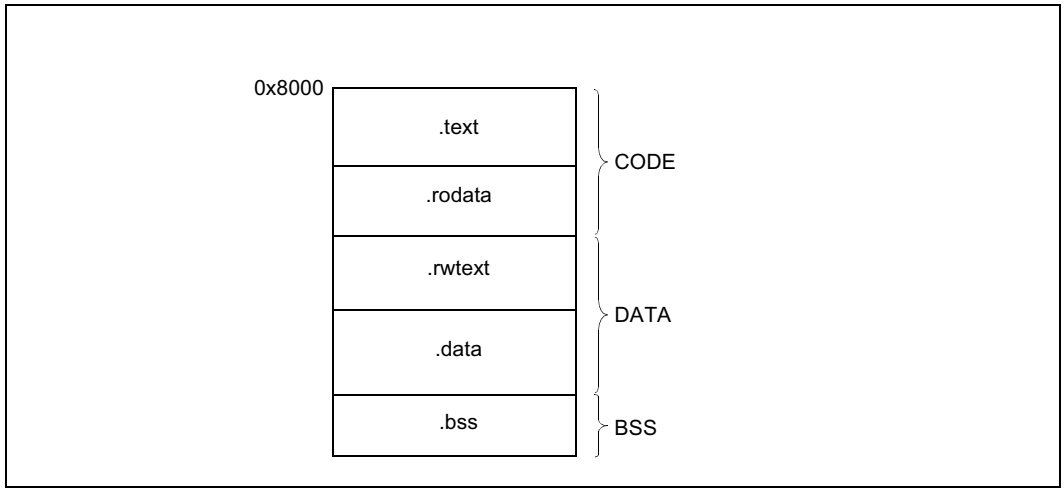
Table 2. Section Names and Associated Segment

Name	Description	Segment	Access
.text	Application code, string constants.	CODE	execute
.rodata	Constant data.	CODE	read
.data	Initialized data.	DATA	read/write
.bss	Uninitialized data.	BSS	read/write

2.6.2 Default Allocation – Default Layout

The compiler and the linker provide a default allocation of segments and sections, as shown below:

Figure 2. Default Allocation of Segments and Sections



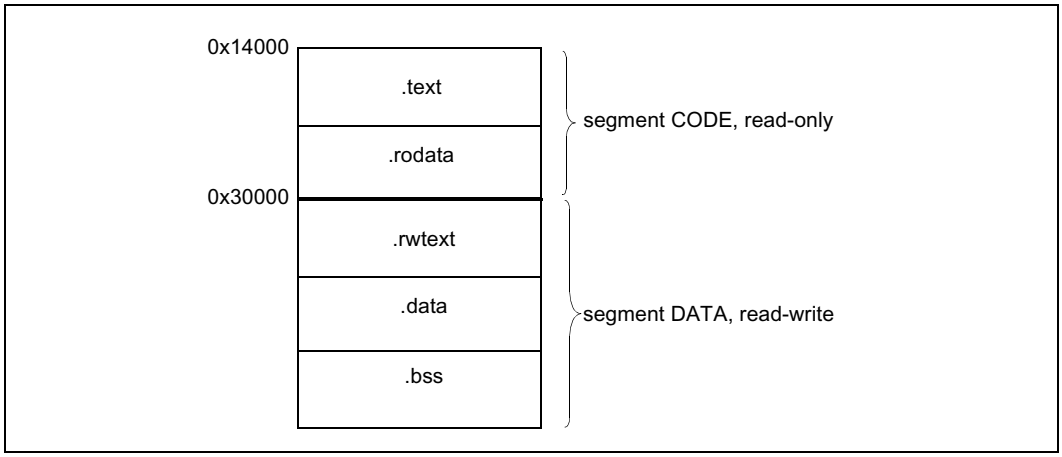
The segments are aligned contiguously within the memory starting at the default base address 0x8000, with the sections also being aligned contiguously within the segment. The sections `.text` and `.rodata` are aligned in the CODE segment by default. The sections `.rwtext` and `.data` are aligned in the DATA segment. The `.bss` section is aligned in the BSS segment. The default alignment of sections within a segment is 4 bytes. You can change the alignment with the linker command `LOAD` described in Section 4.2.2.3, “Layout Body” on page 88.

Example 3. Changing the Alignment of Sections

The code below is the fragment of a build file:

```
...
...
LAYOUT
    SEGMENT CODE BASE 0x14000
            LOAD SECTION .text
            SECTION .rodata

    SEGMENT DATA BASE 0x30000
            LOAD SECTION .rwtext
            SECTION .data
            SECTION .bss
```



The layout of the output file defines two segments. Segment CODE starts at a base address of 0x14000. Segment DATA starts at a base address of 0x30000. Segment CODE contains the sections `.text` and `.rodata`. Section `.text` starts at 0x14000 followed by section `.rodata`. Segment DATA contains the sections `.rwtext`, `.data`, and `.bss`. Section `.rwtext` starts at the base address of segment DATA followed by the remaining sections.

Options 3

This chapter describes all command line options of the Intel® Linker, including their syntax and examples.

The chapter contains the following sections:

- Section 3.1, “Option Summary” on page 26 covers a summary of all command line options. The options are sorted according to their effect.
- Section 3.2 - Section 3.36 provide a detailed description of every option, which are sorted in alphabetical order.

3.1 Option Summary

This section provides a summary of available options, sorted according to their effects. An alphabetical overview is provided in Section 2.3, "Overview of Options" on page 17. A detailed description of each option follows in the remaining sections.

Input File	
Option	**Description**
`-buildfile`	Processes the default build file. The default build file name is the base name of the output file with the extension .bld.
`-buildfile=`*`name`*	Processes the specified build file *`name`*.
`-buildmacro` *`macroname`*`[=`*`text`*`]`	Enables usage of macros inside the build file.
`-keep` *`arg`*`{,`*`arg`*`}`	Specifies which sections are not to be removed by the -remove option. This option selectively overrides the -remove option.
`-remove[=`*`arg`*`{,`*`arg`*`}]`	Removes all non-referenced input sections, of types specified, or all types if none specified.

Listing File	
Option	**Description**
`-listing`	Creates a listing file.
`-listing=`*`name`*	Creates a listing file with name *`name`*.
`-info=`*`arg`*`{,`*`arg`*`}`	Specifies parameters for the listing file. Available values for *`arg`* are: `errors`, `segments`, `sections`, `modules`, `memory`, `symbols`, `map`, and `buildfile`.

Command Line	
Option	**Description**
`@filename`	Specifies an option file. See Section 2.1.2, "Option Arguments" on page 14
`-help`	Prints a help list on the screen.

C++	
Option	**Description**
`-cdtorseg`	Changes the default allocation of constructor and destructor tables. This option is only valid if `-cplus` is set.
`-cplus`	Creates tables for constructors and destructors of static C++ objects.
`-ctorpattern`	Specifies the pattern for constructor callers. This option is only valid if `-cplus` is set.
`-ctortab`	Specifies the symbol name of the constructor table. This option is only valid if the option `-cplus` is set.
`-dtorpattern`	Specifies the pattern for destructor callers. This option is only valid if `-cplus` is set.
`-dtortab`	Specifies the symbol name of the destructor table. This option is only valid if the option `-cplus` is set.

Output	
Option	**Description**
`-output` *name*	Specifies the output filename.
`-entry` *arg*	Defines the entry point. *arg* can be a symbol name or an address.
`-base` *arg*	Sets the base address to the absolute value *arg*. *arg* can be a hexadecimal, decimal, or octal absolute value. Default base address is 0x8000.
`-commalign=`*number*	Selects the global alignment for common data.
`-init` *arg*`{,`*arg*`}`	Initializes the specified segments.
`-export` *arg*`{,`*arg*`}`	Specifies global symbols that are to be written to the final symbol tables. These symbols may be exported to other modules.
`-import` *arg*`{,`*arg*`}`	Specifies global symbols that are to be written to the final symbol tables. These are symbols which require to be imported from other modules.
`-mapcomdat`	Maps global common data to the first input module that is referring to it.
`-nosymbols`	Suppresses the generation of the symbol table.
`-normempty`	Suppresses the removal of empty sections.
`-ropi`	Generates relocation information for position-independent read-only segments.
`-rwpi`	Generates relocation information for position-independent read-write segments.
`-shared`	Generates a shared object file.
`-dynamic`	Causes any shared object file to be generated in the Linux* format.
`-dynamic`	Causes any shared object file to be generated in the format specified by *type*, where *type* can be `Linux` or `ARM`.

`-forceblx`	Changes BL instructions to BLX to avoid generation of veneer code.

Debug Information	
Option	**Description**
`-debug`	Links debug information to the output file.

Information on Link Process	
Option	**Description**
`-nowarnings`	Suppresses warning messages.
`-unref`	Displays warning messages for unreferenced symbols.
`-strict`	Forces strict consistency checking of the linked modules.
`-verbose`	Prints additional information on the link process on the screen. This option is the same as `-verbose=0`.
`-verbose=`*`num`*	Prints additional information on the link process on the screen. Additionally, you can specify by *`num`* which information is to be displayed. Possible values for *`num`* are 0, 1, 2, 3, and 4.

Note: The option names start with a minus sign '-', but this can be replaced with a forward slash '/' .

3.2 @filename

Syntax: `@filename`

Description:
When `@filename` is set, the Intel® Linker opens the file `filename` to read additional command line options.

Further Information:
Section 2.1.2, “Option Arguments” on page 14

3.3 -base

Syntax: `-base arg`

Abbreviation: `-b arg`

Description:
The option `-base` sets the base address *arg* of the default layout, to ensure that the first segment starts at the specified base address. All other segments are loaded contiguously. *arg* is an absolute value and can be specified as an octal, hexadecimal, or decimal number. The base address is set to default value 0x8000 when the option `-base` is not set. Note that this option is overwritten by the base address set in the LAYOUT block inside a build file. The option `-base` cannot be used if the option `-buildfile` is set.

Example 4. Option -base

```
ldxsc -l b.o
```

The command line above allocates the first segment in memory at the default base address 0x8000 as shown in the corresponding listing file.

```
Memory Usage:

Start       End         Section          Segment       Ovrly  Type  Attr

0x00008000  0x00008008  (ROCODE)         CODE                 Code  RO
0x00008008  0x0000802c  (RODATA)         CODE                 Code  RO
0x0000802c  0x00008030  (RWDATA)         DATA                 Code  RW
```

```
ldxsc -l -base 0x30000 b.o
```

This command line allocates the first segment at the base address 0x30000 in memory. The listing file displays the changed base address.

```
Memory Usage:

Start       End         Section          Segment       Ovrly  Type  Attr

0x00030000  0x00030008  (ROCODE)         CODE                 Code  RO
0x00030008  0x0003002c  (RODATA)         CODE                 Code  RO
0x0003002c  0x00030030  (RWDATA)         DATA                 Code  RW
```

Further Information:
Section 3.4, “-buildfile” on page 31
Section 4.2.2.1, “Base Address” on page 85

3.4 -buildfile

Syntax: `-buildfile[=name]`

Abbreviation: `-bf[=name]`

Description:
The option `-buildfile=name` specifies the build file *name* that is processed by the linker. If no extension is specified for *name*, the linker assumes the default extension .bld. The argument `=name` is optional and can be ignored, such as in `-buildfile`. In which case the build file name is assumed to be the base name of the first input object with the extension .bld. The build file must be located in the *current directory*, which is the directory in which the link process is being executed.

Input files can be specified on the command line or in a build file using the directive `_INPUT_`. The following rule applies:

- Where the buildfile does not contain the `_INPUT_` directive, those object files specified on the command line are passed to the linker first. Then object files specified inside the build file are passed to the linker. This order of events occurs even if the buildfile is specified first on the command line.
- If the build file contains the directive `_INPUT_` inside the build file, then the object files are passed to the linker in the order as specified in the build file, until this directive is reached. The directive `_INPUT_` has the effect of causing those object files specified specifically on the command line to be passed to the linker. After which any remaining object files specified in the buildfile after the directive `_INPUT_` are passed to the linker.

Note: Where the optional argument `=name` is used there must be no spaces before and after the "=" sign.

Examples 5. Option -buildfile

Assuming the build file is named b.bld, the following command line processes this build file:

```
ldxsc -bf b.o a.o
```

Assuming the contents of the build file b.bld is as follows:

```
INPUT
   x.o
   y.o
   z.o
```

Then the following command line:

```
ldxsc -bf b.o a.o
```

processes the object files for linking in the following order:

1. b.o
2. a.o
3. x.o
4. y.o
5. z.o

Assuming the contents of the build file is as follows:

```
INPUT
   x.o
   _INPUT_
   y.o
   z.o
```

Then the following command line:

```
ldxsc -bf b.o a.o
```

processes the object files for linking in the following order:

1. x.o
2. b.o
3. a.o
4. y.o
5. z.o

Example 6. Option -buildfile=*name*

The command line shown below:

```
ldxsc -buildfile=testbuild a.o
```

links the object file `a.o` and processes the build file `testbuild.bld`.

The command line shown below:

```
ldxsc -buildfile=testbuild.bf a.o
```

processes the contents of the build file `testbuild.bf` and then links with the object file `a.o`.

Assuming the contents of the build file `testbuild.bld` is as follows:

```
INPUT
   x.o
   y.o
   z.o
```

Then the command line shown below:

```
ldxsc -bf=testbuild b.o a.o
```

processes the object files for linking in the following order:

1. b.o
2. a.o
3. x.o
4. y.o
5. z.o

Example 7. Option -bf=*name*

Assuming the contents of the build file is as follows:

```
INPUT
   x.o
   _INPUT_
   y.o
   z.o
```

Then the command line shown below:

```
ldxsc -bf=testbuild b.o a.o
```

processes the object files for linking in the following order:

1. x.o
2. b.o
3. a.o
4. y.o
5. z.o

Further Information:
Section 3.4, “-buildfile” on page 31
Section 4.2.3, “INPUT Block” on page 91

3.5 -buildmacro

Syntax: `-buildmacro` *`macroname`*`[=`*`text`*`]`

Abbreviation: `-bm` *`macroname`*`[=`*`text`*`]`

Note: There must be no spaces before and after the "=".

Description:
The option `-buildmacro` enables the use of macros inside a build file, where *`macroname`* is the name of the macro inside the build file. The macro is called using the construction `@`*`macroname`*. The optional parameter *`text`* is used to assign the specified macro a value.

Examples 8. Option -buildmacro

Assuming the following build file test.bld that contains the macro `@p1`:

```
layout
    segment CODE
        base 0x8000
        load
            label global __code_start__
            section _ro_
            label global __code_end__
    segment DATA
        base align 0x1000
        load
            label global __data_start__
            section _rw_
            label global __data_end__
    segment BSS
        base align 0x1000
        load
            label global __bss_start__
            section _bss_
            label global @p1
```

Command line:

```
ldxsc -l -o test a.o b.o -bf -bm p1=__bss_end__
```

This sets the macro `p1` to the value `__bss_end__`. This means that the global label "`__bss_end__`" is set in the BSS segment.

Assuming the following build file test.bld that contains the macro `@object1` and `@object2`:

```
input

   @object1
   @object2
```

Command line:

```
ldxsc -l -o test -bf -bm object1=a.o -bm object2=b.o
```

This sets the macro `object1` to the file a.o and macro `object2` to the file b.o. Both files are linked to the object file.

Further Information:
Section 3.4, “-buildfile” on page 31

3.6 -cdtorseg

Syntax: `-cdtorseg` *arg*

Abbreviation: `-cds` *arg*

Note: This option requires the further option `-cplus` to be set, and is valid for the default layout only.

Description:
The option `-cdtorseg` changes the default allocation of constructor and destructor tables. By default, they are allocated in the section `.rodata` inside the CODE segment. The argument *arg* specifies the segment the tables will be stored.

Example 9. Option -cdtorseg

```
ldxsc -l -cplus a.o
```

The command line above links the input file a.o and creates tables for constructors and destructors. By default, the tables are stored in the CODE segment as shown in the listing file (fragment) below.

```
C++ constructor

    Section                 Base        Size        Align  Segment
    .rodata                 0x0000802c  0x00000004      4  CODE

C++ destructor

    Section                 Base        Size        Align  Segment
    .rodata                 0x00008030  0x00000004      4  CODE
```

```
ldxdc -l -cplus -cds DATA a.o
```

The allocation of the tables can be changed when using the option `-cds DATA` for example. These tables are now stored in the DATA segment as shown in the example below.

```
C++ constructor

    Section                 Base        Size        Align  Segment
    .rodata                 0x00008030  0x00000004      4  DATA

C++ destructor

    Section                 Base        Size        Align  Segment
    .rodata                 0x00008034  0x00000004      4  DATA
```

Further Information:
Section 2.6, “Allocation of Segments and Sections” on page 22
Section 3.7, “-cplus” on page 38

3.7 -cplus

Syntax: `-cplus`

Abbreviation: `-cp`

Description:
The option `-cplus` enables the generation of constructor and destructor tables of static C++ objects. By default, the following values are set:

Symbol	Default Value
pattern of constructor callers	`__sti__*`
pattern of destructor callers	`__std__*`
symbol of constructor table	`__ctors`
symbol of destructor table	`__dtors`

If this option is specified, the options `-cdtorseg`, `-ctorpattern`, `-ctortab`, `-dtorpattern`, and `-dtortab` can also be passed to the linker to modify the defaults. Additionally, the listing file contains the section "C++ Definitions" displaying information on C++.

Example 10. Option -cplus

```
-ldxsc -listing -cplus a.o
```

The command line above creates the constructor and destructor tables. When a listing file is also generated, the location of the tables within the segment is displayed.

```
C++ constructor

    Section                   Base        Size        Align  Segment
    .rodata                   0x0000802c  0x00000004      4  CODE

C++ destructor

    Section                   Base        Size        Align  Segment
    .rodata                   0x00008030  0x00000004      4  CODE
...
...
C++ Definitions:

    Constructor Symbols: __sti__* (count=0)

    Destructor  Symbols: __std__* (count=0)

    Constructor Table  : __ctors at 0x0000802c

    Destructor  Table  : __dtors at 0x00008030
...
```

Further Information:

Section 3.6, “-cdtorseg” on page 37
Section 3.9, “-ctorpattern” on page 41
Section 3.10, “-ctortab” on page 42
Section 3.12, “-dtorpattern” on page 44
Section 3.13, “-dtortab” on page 45
Section 5.2.9, “C++ Definitions” on page 108

3.8 -commalign

Syntax: `-commalign=`*`number`*

Abbreviation: `-ca=`*`number`*

Description:
With the option `-commalign` the global alignment for common data is selected. Possible values for *`number`* are `4`, `8`, `16`, `32`, … and the special value `natural`. If `natural` is selected, the alignment is determined by the common data item with the largest size. The alignment in this case would be computed to be smallest two's complement greater or equal that size.

Example 11. Option -commalign

```
-commalign=8
-ca=natural
```

3.9 -ctorpattern

Syntax: `-ctorpattern` *arg*

Abbreviation: `-ctp` *arg*

Note: This option requires option `-cplus`

Description:
The option `-ctorpattern` specifies the pattern for functions which are assumed to be constructor callers of static C++ objects. The default pattern for these functions is `__sti__*`.

Example 12. Option -ctorpattern

```
ldxsc -l -cplus a.o
```

The command line above creates the default pattern for constructor callers of static C++ objects as shown in the listing file.

```
C++ Definitions:

    Constructor Symbols: __sti__* (count=0)

    Destructor  Symbols: __std__* (count=0)

    Constructor Table  : __ctors at 0x0000802c

    Destructor  Table  : __dtors at 0x00008030
```

```
ldxsc -l -cplus -ctorpattern pattern a.o
```

This command line creates the pattern "`pattern`" for constructor callers as shown in the listing file.

```
C++ Definitions:

    Constructor Symbols: pattern* (count=0)

    Destructor  Symbols: __std__* (count=0)

    Constructor Table  : __ctors at 0x0000802c

    Destructor  Table  : __dtors at 0x00008030
```

Further Information:
Section 3.7, "-cplus" on page 38

3.10 -ctortab

Syntax: `-ctortab` *arg*

Abbreviation: `-ctt` *arg*

Note: This option requires option `-cplus`

Description:
The option `-ctortab` specifies the name of the public symbol which is defined at the start of the constructor table. The default symbol name is `__ctors`.

Example 13. Option -ctortab

```
ldxsc -l -cplus a.o
```

The command line above creates the default symbol name for the constructor table as shown in the listing file.

```
C++ Definitions:

    Constructor Symbols: __sti__* (count=0)

    Destructor  Symbols: __std__* (count=0)

    Constructor Table  : __ctors at 0x0000802c

    Destructor  Table  : __dtors at 0x00008030
```

```
ldxsc -l -cplus -ctortab newname a.o
```

This command line creates the symbol name `newname` for the constructor table as shown in the listing file.

```
C++ Definitions:

    Constructor Symbols: __sti__* (count=0)

    Destructor  Symbols: __std__* (count=0)

    Constructor Table  : newname at 0x0000802c

    Destructor  Table  : __dtors at 0x00008030
```

Further Information:
Section 3.7, “-cplus” on page 38

3.11 -debug

Syntax: `-debug`

Abbreviation: `-db`

Description:
The option `-debug` links debug information to the object file. By default, if this option is not set, debug information is not generated.

To debug the output file of the linker, the Intel® XDB Debugger needs a boundfile, which needs to be generated using the Intel® DWARF2BD Object Converter.

Further Information:
Intel® Object Converters Manual
Intel® XDB Debugger Reference Manual

3.12 -dtorpattern

Syntax: `-dtorpattern` *arg*

Abbreviation: `-dtp` *arg*

Note: This option requires option `-cplus`

Description:
The option `-dtorpattern` specifies the pattern for functions which are assumed to be destructor callers of static C++ objects. The default pattern for these functions is `__std__*`.

Example 14. Option -dtorpattern

```
ldxsc -l -cplus a.o
```

The command line above creates the default pattern for destructor callers of static C++ objects as shown in the listing file.

```
C++ Definitions:

    Constructor Symbols: __sti__* (count=0)

    Destructor  Symbols: __std__* (count=0)

    Constructor Table  : __ctors at 0x0000802c

    Destructor  Table  : __dtors at 0x00008030
```

```
ldxsc -l -cplus -dtorpattern pattern a.o
```

This command line creates the pattern "`pattern`" for destructor callers as shown in the listing file.

```
C++ Definitions:

    Constructor Symbols: __sti__* (count=0)

    Destructor  Symbols: pattern* (count=0)

    Constructor Table  : __ctors at 0x0000802c

    Destructor  Table  : __dtors at 0x00008030
```

Further Information:
Section 3.7, "-cplus" on page 38

3.13 -dtortab

Syntax: `-dtortab` *arg*

Abbreviation: `-dtt` *arg*

Note: This option requires option `-cplus`

Description:
The option `-dtortab` specifies the name of the public symbol which is defined at the start of the destructor table. The default symbol name is `__dtors`.

Example 15. Option -dtortab

```
ldxsc -l -cplus a.o
```

The command line above creates the default symbol name for the destructor table as shown in the listing file.

```
C++ Definitions:

    Constructor Symbols: __sti__* (count=0)

    Destructor  Symbols: __std__* (count=0)

    Constructor Table  : __ctors at 0x0000802c

    Destructor  Table  : __dtors at 0x00008030
```

```
ldxsc -l -cplus -dtortab newname a.o
```

This command line creates the symbol name `newname` for the destructor table as shown in the listing file.

```
C++ Definitions:

    Constructor Symbols: __sti__* (count=0)

    Destructor  Symbols: __std__* (count=0)

    Constructor Table  : __ctors at 0x0000802c

    Destructor  Table  : newname at 0x00008030
```

Further Information:
Section 3.7, “-cplus” on page 38

3.14 -dynamic

Syntax: `-dynamic[=type]`

Abbreviation: `-dyn[=type]`

Description:
When the option `-dynamic` is set, the linker generates extra sections in the resulting executable to hold dynamic data. Setting this option as `-dynamic=Linux` is analogous to setting the option as `-dynamic` (without any argument). The option `-dynamic=Linux` causes the linker to generate a dynamic file in Linux Elf format. Setting the option as `-dynamic=ARM` causes the linker to generate a dynamic file in ARM Elf format. When the additional option `-shared` is set the linker generates a file of the Elf format type ET_DYN (dynamic library form). Without the option `-shared` being set the linker generates a file of the Elf format type ET_EXEC (executable with dynamic information form).

The argument *type* is optional and is used to specify whether ARM* or Linux* type formatting is used in the dynamic executable. If omitted as in `-dynamic`, the default Linux* type formatting is assumed.

type	Description
ARM	Produces dynamic executable using ARM* format.
Linux	Produces dynamic executable using Linux* format, (default).

Note: Where the optional argument `=type` is used there must be no spaces before and after the "=" sign.

Further Information:
Section 3.16, "-export" on page 48
Section 3.19, "-import" on page 52
Section 3.33, "-shared" on page 67

3.15 -entry

Syntax: `-entry` *arg*

Abbreviation: `-e` *arg*

Description:
The option `-entry` defines the entry point of the application. The address of the entry point is specified by *arg*. *arg* is either defined by a symbol name or directly by an address value. The default value of the entry point is 0x8000. This option is useful if the entry point is not defined in an input module.

Example 16. Option -entry

```
ldxsc -l a.o
```

The command line above links the object file a.o. Since no entry point is defined in the input module a.o, the default value 0x8000 is assumed. The listing file displays the information.

```
Symbol Table:

Entry Point: No entry point, 0x8000 defaulted
```

```
ldxsc -l -entry 0xFFF0 a.o
```

This command line sets the entry point of the application to the value 0xFFF0 as shown in the listing file below.

```
Symbol Table:

Entry Point: 0x0000fff0
```

3.16 -export

Syntax: `-export arg{,arg}`

Abbreviation: `-ex arg{,arg}`

Description:
The option `-export` has two different meanings depending on whether `-dynamic=ARM` is used or not. This option `-export` enables the selection of one or more symbols, specified in the comma separated list *`arg, arg`*,..., to be written to the conventional static symbol table or any existing dynamic symbol table. These symbols then become available, that is they can be exported, to other modules. A dynamic symbol table is only generated when the `-dynamic=ARM` option is used.

By default, all global symbols are held in the static symbol table, making them all available for use by other modules. The option `-export *` has the same effect. All global symbols held in the static symbol table are also placed in the dynamic symbol table.

To stop all global symbols from being written to the symbol tables this option can be used with a nonexistent symbol name, such as for example `-export _none_`.

This command is useful for hiding global symbols for other tools, such as the debugger or the linker if this file is used as a symbol file in another link process.

Note: `-export` does not affect the symbol table inside the listing file, but only the symbol tables of the output file.

Examples 17. Option -export

```
ldxsc -l a.o b.o -ex caller
```

The command line above exports only the symbol `caller` to the symbol table in the output file.

```
ldxsc -l a.o b.o -ex caller,foo
```

This command line exports the symbols `caller` and `foo` to the symbol table in the output file.

Further Information:
Section 3.14, “-dynamic” on page 46
Section 3.19, “-import” on page 52
Section 3.33, “-shared” on page 67

3.17 -forceblx

Syntax: `-forceblx`

Abbreviation: `-fblx`

Description:
The option `-forceblx` forces the Intel® Linker to check for branches that would change the processor mode from ARM* to Thumb* or vice versa. In such a case, the linker patches the branch instruction to BLX to avoid the generation of veneer code.

Setting this option can increase the performance of the application.

3.18 -help

Syntax: `-help`

Abbreviation: `-?`

Description:
When `-help` is set, a summary of all available options is printed on the screen. The summary contains a short description of each option.

Example 18. Option -help

```
ldxsc -help
```

prints the following output:

```
Intel(R) Linker for Intel(R) XScale(TM) Microarchitecture, Version 1.1.12
Copyright (C) 2001-2003 Intel Corporation.
Portions copyright (c) 1982-1994 Kinetech, Inc. [or its assignee].
All rights reserved.

Usage: ldxsc [<options>] {objects}
[-(output|o) <arg>]        specify the output file name (default: l.x)
[-(buildfile|bf)]          process build file
[-(buildfile=|bf=)<arg>]   process build file named <arg>
[-(base|b) <arg>]          define the base address when using default layout
                           (<arg> must be an absolute value)
[-(listing|l)]             generate listing file
[-(listing=|l=)<arg>]      generate listing file named <arg>
[-(info=|i=)<arg,...>]     set listing options
   [all]                   print all available info
   [buildfile]             print contents of build file
   [errors]                print diagnostic messages
   [map]                   print the symbol table
   [memory]                print the memory usage
   [modules]               print the module summary
   [sections]              print the section summary
   [segments]              print the segment summary
   [symbols]               print module and segment symbols
[-(debug|db)]              create debug information
[-(nosymbols|ns)]          don't generate symbol table
[-(nowarnings|nw)]         suppress warning messages
{-(buildmacro|bm) <arg>}   specify a build file macro (<arg>: macname[=text])
[-(entry|e) <arg>]         define the entry point (<arg> can be a symbol name
                           or an absolute value)
[-(unref|ur)]              display warning messages for unreferenced symbols
[-(export|ex) <arg,...>]   specify symbols of symbol table, <arg> is '*' or a
                           comma-separated list of symbols
[-ropi]                    generate relocation info for position independent RO
                           segments
[-rwpi]                    generate relocation info for position independent RW
                           segments
[-(init|in) <arg,...>]     initializes one or more segments, <arg> is comma
                           separated list of segment names
```

```
[-(cplus|cp)]              generate tables for con/destructors of static C++
                           objects
[-(cdtorseg|cds) <arg>]    name of the segment where the con/destructor tables
                           are stored (only valid, if '-cplus' is set and
                           default layout is used)
[-(ctorpattern|ctp) <arg>] pattern for constructor initializer
[-(dtorpattern|dtp) <arg>] pattern for destructor initializer
[-(ctortab|ctt) <arg>]     symbol name of the constructor table (only valid, if
                           '-cplus' is set)
[-(dtortab|dtt) <arg>]     symbol name of the destructor table (only valid, if
                           '-cplus' is set)
[-(forceblx|fblx)]         patch BL to BLX instructions to avoid veneer
                           generation
[-(strict|sc)]             force strict checking of processed input modules
[-(normempty|nre)]         don't remove empty sections
[-(commalign=|ca=)<num>]   set alignment for common symbols (<num> =
                           4,8,16,...,natural)
[-(mapcomdat|mcd)]         map common data allocation to input modules
[-(shared|sh)]             create a shared library
[-(remove|rm)]             remove unused sections
[-(verbose|v)]             provide verbose messages (default level: 0)
[-(verbose=|v=)<num>]      set verbose level (0 <= level <= 4)
[-(help|?)]                display this usage
```

3.19 -import

Syntax: `-import arg{,arg}`

Abbreviation: `-im arg{,arg}`

Description:
The option `-import` enables one or more global symbols, specified in the comma separated list *arg, arg*,..., to be selected for import from another module. These symbols are placed within the dynamic symbol table, and are resolved from other external modules at load and run time. They are imported from external modules even if they could be resolved internally within the module being linked. This allows a module segment to be optionally replaced with a similarly named segment from another module.

A dynamic symbol table is only generated if the `-dynamic=ARM` option is used, otherwise the `-import` option is ignored.

To stop all of the global symbols from being imported and written to the dynamic symbol table, this option can be used with a nonexistent symbol name, such as for example `-import _none_`.

Note: Any unresolved symbols residing in a segment marked as DYNAMIC in the buildfile and having relocation types R_ARM_ABS12 and R_ARM_THM_ABS5 are automatically placed within the dynamic relocation table.

Examples 19. Option -import

```
ldxsc -l a.o b.o -im caller,shout,hail
```

The command line above places the symbols `caller`, `shout` and `hail` into the dynamic symbol table for import from other modules at load time.

Further Information:
Section 3.14, “-dynamic” on page 46
Section 3.16, “-export” on page 48
Section 3.33, “-shared” on page 67

3.20 -info

Syntax: `-info=arg{,arg}`

Abbreviation: `-i=arg{,arg}`

Note: This option requires option `-listing` or `-listing=name`. Otherwise it has no effect.

Description:
The option `-info` specifies parameters for the listing file. By default, the listing file contains all supported sections. The parameter *`arg`* specifies which section is written to the listing file. At least one parameter must be specified; if more than one parameter is specified, they must be separated by commas. The header section of the listing file is always created. The parameter *`arg`* may have the following values:

arg	Description
`all`	Is identical to setting all parameters mentioned in this table.
`buildfile`	Prints the contents of the build file in the listing file. See also Section 5.2.10, “Build File” on page 109.
`errors`	Prints diagnostic messages inside the header in the listing file. See also Section 5.2.1, “Header” on page 99.
`map`	Prints the symbol table in the listing file. See also Section 5.2.8, “Symbol Table” on page 106.
`memory`	Prints the memory usage in the listing file. See also Section 5.2.5, “Memory Usage” on page 103.
`modules`	Prints the module summary in the listing file. See also Section 5.2.4, “Module Summary” on page 102.
`sections`	Prints the segment-section summary in the listing file. See also Section 5.2.3, “Segment-Section Summary” on page 101.
`segments`	Prints the segment summary in the listing file. See also Section 5.2.2, “Segment Summary” on page 100.
`symbols`	Prints the listing of module symbols and segment symbols in the listing file. See also Section 5.2.6, “Listing of Module Symbols” on page 104 and Section 5.2.7, “Listing of Segment Symbols” on page 105.

Example 20. Option -info

```
ldxsc -listing a.o
```

The command file above generates a listing file a.map that contains all sections: header, segment, section, and module summary, memory usage, listing of module and segment symbols, symbol table, and contents of build file.

```
ldxsc -listing -info=modules,symbols,map a.o
```

This command file prints the module summary, the module and segment symbols, and the symbol table in the listing file.

Further Information:
Section 3.7, “-cplus” on page 38
Section 3.23, “-listing” on page 57
Chapter 5, “Listing File”

3.21 -init

Syntax: `-init` *arg*{,*arg*}

Abbreviation: `-in` *arg*{,*arg*}

Description:
The option `-init` forces the initialization of the specified segments during the set-up of the run-time environment. The names of the sections are listed in a comma separated list *arg,arg,*... The contents of all specified segments are stored to a section named ‘init_data’, which again is linked by default to the CODE segment. After loading this segment to target memory, the contents of the initialized segments are copied to the locations to which they were linked to. This enables dynamic initialization of a program that, for example, resides in ROM memory.

Further Information:
Section 3.31, “-ropi” on page 65
Section 3.32, “-rwpi” on page 66

3.22 -keep

Syntax: `-keep` *arg{,arg}*

Description:
The option `-keep` partially overrides the `-remove` option by specifying sections not to be removed. The names of the sections are listed in a comma separated list *arg,arg,*...
This option can only be used in conjunction with the `-remove` option, otherwise it will be ignored.

If the option `-remove` is set, the linker removes all non-referenced input sections. Using the `-keep` option forces those sections listed to be kept within the final image even if they are non-referenced input sections. Such sections may, for example, be used for test purposes.

Further Information:
Section 3.30, “-remove” on page 64

3.23 -listing

Syntax: `-listing[=`*`name`*`]`

Abbreviation: `-l[=`*`name`*`]`

Description:
The option `-listing` creates a listing file named *`name`*. If no extension is specified, the default extension of the listing file is .map. The argument *`name`* is optional and may be omitted, as in `-listing`, in which case a default listing file name is assumed as follows. Should the option `-output` be used without specifying an output file name, then the name of the listing file consists of the base name of the first input object and the extension .map. If the option `-output` also specifies an output file name, then the name of the listing file is the base name of the output file with extension .map. The created listing file contains all sections as described in Section 5.2, "Contents of a Listing File" on page 98.

Note: Where the optional argument `=`*`name`* is used there must be no spaces before and after the "=" sign.

Description:

Examples 21. Option -listing

```
ldxsc -listing a.o b.o
```

The generated listing file is **a.map**.

```
ldxsc -listing -o test a.o
```

The generated listing file is **test.map**.

Examples 22. Option -listing=*name*

```
ldxsc -listing=list a.o
```

The generated listing file is **list.map**.

```
ldxsc -listing=list.lst a.o
```

The generated listing file is **list.lst**.

Further Information:
Section 3.7, "-cplus" on page 38
Section 3.23, "-listing" on page 57
Section 3.20, "-info" on page 53
Chapter 5, "Listing File"

3.24 -mapcomdat

Syntax: `-mapcomdat`

Abbreviation: `-mcd`

Description:
The option `-mapcomdat` maps global common data to the first input module that is referring to it. The data will be allocated in a section named “comdat” which is added to the section list of this module. Using this option together with the linker command `LOAD SECTION comdata OF` *`module`* enables you to split the allocation of common data over several output segments.

Further Information:
Section 4.2.2.3, “Layout Body” on page 88

3.25 -noinfo

Syntax: `-noinfo`

Abbreviation: `-ni`

Description:
Setting this option suppresses the generation of information messages. By default, this option is not set, and information messages are being generated.

Note: The option `-noinfo` is independent of the options `-info` and `-verbose`.

Further Information:
Section 3.20, “-info” on page 53
Section 3.36, “-verbose” on page 70

3.26 -normempty

Syntax: `-normempty`

Abbreviation: `-nre`

Description:
The option `-normempty` causes the Intel® Linker not to remove empty sections; they are linked to the output instead. Should empty sections be used to define global symbols, this option is useful to keep them. By default, the Intel® Linker removes empty sections.

Note: Setting the option `-remove` implicitly sets the option `-normempty`. That is, with the activated `-remove` option the linker removes empty segments only when they are not referenced.

Further Information:
Section 3.30, “-remove” on page 64

3.27 -nosymbols

Syntax: `-nosymbols`

Abbreviation: `-ns`

Description:
The option `-nosymbols` suppresses the generation of the symbol table in the object file and listing file. This is useful to save disk space.

Example 23. Option -nosymbols

```
ldxsc -l -nosymbols a.o b.o
```

When the command line above is invoked, the symbol table is omitted in the object file as well as in the listing file. The “Mapping Symbols” and “Global Symbols” section is empty.

```
Symbol Table:

Entry Point: No entry point, 0x8000 defaulted

--------------------------------------------------------------------------------

Mapping Symbols:

Segment       Address     Symbol


--------------------------------------------------------------------------------

Global Symbols:

Segment       Section           Address     Size        Type  Symbol
```

3.28 -nowarnings

Syntax: `-nowarnings`

Abbreviation: `-nw`

Description:
The option `-nowarnings` turns off the printing of warning messages to the screen, and also to the listing file if one is being generated.

Example 24. Option -nowarnings

```
ldxsc a.o
```

Using the command line above produces the following output on the screen:

```
LDXSC-W-WARNING:666: following symbols are undefined:
LDXSC-W-WARNING:667: a.o : symbol mylib: weak
```

```
ldxsc -nowarnings a.o
```

This command line suppresses the display of warning messages on the screen.

3.29 -output

Syntax: `-output` *name*

Abbreviation: `-o` *name*

Description:
The option `-output` specifies the name *name* of the output file. If the option `-output` is not used, the output filename is the base name of the first input file, with the extension changed to .x.

Examples 25. Option -output

```
ldxsc a.o b.o
```

The command line above generates the linker output filename **a.x**.

```
ldxsc -o test a.o b.o
```

This command line generates the linker output filename **test.x**.

```
ldxsc -o test.out a.o b.o
```

This command line generates the linker output filename **test.out**.

3.30 -remove

Syntax: `-remove[=arg,{arg}]`

Abbreviation: `-rm[=arg,{arg}]`

Note: If the optional comma separated type list is used, there must be no spaces before or after the = sign.

Description:
If the option `-remove` is set, the linker removes all non-referenced input sections of the types having attributes specified in the optional type list. The types in the list can be:

Type	Description
RW	Remove all read write sections which are not referenced.
RO	Remove all read-only sections which are not referenced.
ZI	Remove all Zero Initialization sections which are not referenced (the .bss sections).

If no optional type list is given, all sections, of all types, are considered for removal.

Input sections of the specified types stay in the final image only if they contain an entry-point, or if they are referred to, directly or indirectly, by a non-weak reference from an input section that contains an entry-point. If neither an input section contains an entry-point, nor an entry-point is specified with the related command line option, this option will be ignored.

The `-keep` option can be used to exclude specific sections for consideration of removal by this option. Using the `-keep` option allows non-referenced sections to be kept in the final image which, for example, can be used for testing purposes.

Note: Setting the option `-remove` implicitly sets the option `-normempty`. That is, with the activated `-remove` option the linker removes empty segments only when they are not referenced.

Examples 26. Option -remove

```
ldxsc -rm=RO a.o b.o
```

The command line above generates the linker output filename **a.x**. with all non-referenced input sections of the type read-only removed.

Further Information:
Section 3.22, “-keep” on page 56
Section 3.26, “-normempty” on page 60

3.31 -ropi

Syntax: `-ropi`

Abbreviation: `(none)`

Description:
If the option `-ropi` is set, all segments with read-only attribute (usually containing code and read-only data) are made position-independent. This means that these segments can be relocated to any valid address on the target hardware, regardless to which addresses they are allocated by the Intel® Linker.

Further Information:
Section 3.21, “-init” on page 55
Section 3.32, “-rwpi” on page 66

3.32 -rwpi

Syntax: `-rwpi`

Abbreviation: `(none)`

Description:
If the option `-rwpi` is set, all segments with read-write attribute (usually containing initialized and uninitialized data) are made position-independent. This means that these segments can be relocated to any valid address on the target hardware, regardless to which addresses they are allocated by the Intel® Linker.

Further Information:
Section 3.21, “-init” on page 55
Section 3.31, “-ropi” on page 65

3.33 -shared

Syntax: `-shared`

Abbreviation: `-sh`

Description:
The option `-shared` is used to specify the type of dynamic executable, and is used in conjunction with the option `-dynamic`. This option is ignored if the option `-dynamic` is not set. When `-shared` is set the linker generates an executable file of the Elf format type ET_DYN (dynamic library). Without `-shared` being set the linker generates a file of the Elf format type ET_EXEC (executable with dynamic information).

Further Information:
Section 3.14, “-dynamic” on page 46
Section 3.16, “-export” on page 48
Section 3.19, “-import” on page 52

3.34 -strict

Syntax: `-strict`

Abbreviation: `-sc`

Description:
The option `-strict` forces strict checking of the consistency of the linked modules. If inconsistencies (like duplicate symbol definitions in different modules or undefined symbols) are found, the Intel® Linker generates an appropriate error message.

Further Information:
Chapter 6, “Diagnostic Messages”

3.35 -unref

Syntax: `-unref`

Abbreviation: `-ur`

Description:
The option `-unref` causes the Intel® Linker to issue a warning message to identify unused symbols if a symbol is defined but never referenced.

Example 27. Option -unref

```
ldxsc -ur -l -o test a.o b.o
```

The command line above issues the following warning messages because the object files contain unreferenced symbols:

```
LDXSC-W-WARNING[665]:caller.o:symbol caller: unreferenced
LDXSC-W-WARNING[665]:caller.o:common symbol myArray: unreferenced
LDXSC-W-WARNING[665]:foo.o:symbol myUnref: unreferenced
```

3.36 -verbose

Syntax: `-verbose[=num]`

Abbreviation: `-v[=num]`

Description:
The option `-verbose` prints messages about linker actions to the screen. The optional argument *num* controls the amount of output, and may have one of the following values:

Value	Description
0	This is the default value. If it is specified, names of passes are printed to the screen. This has the same effect as the option `-verbose`.
1	Prints names of passes and filenames to the screen.
2	Prints names of passes, filenames, and names of sub-passes to the screen.
3	This setting, in additionally to those activities where the value is 2, prints module names in which common sections are checked. It also displays information on constructor and destructor scanning.
4	Additionally to `-verbose=3`, it prints information on handling of common symbols.

If the optional argument *num* is not used, as in `-verbose`, then the default value of *num=0* is assumed.

Note: Where the optional argument *num* is used there must be no spaces before and after the "=" sign.

Example 28. Option -verbose

```
ldxdc -v a.o b.o
```

The command line above invokes the linker as follows:

```
LDXSC: PASS 1 : layout ...
LDXSC: PASS 2 : load and relocate ...
LDXSC: PASS 3 : write output ...
```

Example 29. Option -verbose=*num*

```
ldxsc -verbose=1
```

The command line above generates the following additional output to the screen:

```
LDXSC:        : a.o
LDXSC:        : b.o
LDXSC: PASS 1 : layout ...
LDXSC: PASS 2 : load and relocate ...
LDXSC:        : a.o
LDXSC:        : b.o
LDXSC:        : ZISection$$Table
LDXSC: PASS 3 : write output ...
```

Further Information:
Section 3.23, “-listing” on page 57

This page intentionally left blank.

The Build File 4

This chapter describes the use of a build file, which can be used to define specific settings for alignments of segments and sections, the layout of your linker output file, or to pass additional object files to the linker. See Section 2.6, “Allocation of Segments and Sections” on page 22

A buildfile is only processed by the linker if the option `-buildfile` is specified on the command line.

The chapter contains the following sections:

- Section 4.1, “Lexical Conventions” on page 74 describes the lexical conventions that apply to build files.
- Section 4.2, “Build File Syntax” on page 78 describes the syntax of a build file, which consists of a maximum of 3 optional blocks.

4.1 Lexical Conventions

This section describes the Lexical processes and conventions used by the linker.

4.1.1 The Lexical Process

The first pass of the Intel® Linker scans the lines of the linker input, carrying out syntax and semantic checks. The syntax check performs a lexical analysis, converting the input file into a string of tokens. The semantic check tests the grammar of the input as well as the specified conditions.

4.1.2 Keywords

Keywords are directives such as `ABSOLUTE`, `SECTION`, `ALIGN`, `SEGMENT`. The following rules apply to keywords:

- Keywords are not case-sensitive, i.e. they may be written in upper case or lower case.
- Keyword abbreviations may be used, and are obtained by omitting characters from the end as long as the abbreviations are unambiguous. For example `sec`, `sect`, and `secti` are valid instances of the keyword `section`.

  If an abbreviation is ambiguous, the Intel® Linker issues the following syntax error message:

```
LDXSC-E-ERROR[387]:b.bld:syntax error near ^NPUT^
```

- Names e.g. of variables can be the same as a keyword providing it is 'escaped'. That is by preceding them with the "^"-character. For keywords the Intel® Linker recognizes the leading "^"-character and skips it, accepting the remaining string as the variable name. Where the "^" character is used before a none keyword it becomes part of the symbol name.

Example 30. Escaping Variables

```
^DEFAULT
```

The "^" is skipped and the `DEFAULT` is read as a name and not as keyword.

4.1.3 Names

Names are strings of the following characters:

Table 3. Lexical Conventions – Names

Character	Comment
a-z	Lower case alphabetical.
A-Z	Upper case alphabetical.
0-9	Not in first position.
/	Forward slash.
\	Backslash.
.	Dot (period).
_	Underscore.
:	Colon.
;	Semicolon.
$	Dollar sign.
+	Plus sign.
-	Minus sign.
[	Left bracket.
]	Right bracket.

White space characters such as blank, tab, or newline in any order and number are used to separate the input words from each other. In some cases, special characters are required to isolate members of option lists.

4.1.4 Numbers and Values

Numbers and values are digit strings with a prefix for defining the base and a suffix for defining a factor. All numbers are treated as unsigned values.

Table 4. Lexical Conventions – Prefix Strings

Prefix String	Base	Permitted Numerals
0x	Hexadecimal value.	0-9, a-f, A-F
0	Octal value.	0-7
<none>	Decimal value.	0-9

Table 5. Lexical Conventions – Suffix Strings

Suffix String	Stands for	Factor
K	kilo	Value multiplied by 1024.
P	page	Value multiplied by PAGE size.
<none>		No multiplication.

4.1.5 Expressions

The Intel® Linker supports arithmetic expressions for defining values. An expression must be enclosed in "{" and "}" characters.

Expressions may contain symbol names, constant values, as well as binary and unary operators. The standard C operator precedence rules are applied for evaluation, and parentheses are permitted to change the order of evaluation.

If a symbol value is used, the symbol must be defined by a suitable directive beforehand or in any of the input modules. All computations are performed with unsigned 32-bit values.

The following table lists the available operators, with higher precedence operators first:

Table 6. Lexical Conventions – Operators

Operator	Description
unary -	Negate.
unary `	Bitwise invert.
unary !	Logical not.
unary +	Ignored.
*, /, %	Multiply, divide, modulo.
+, -	Add, subtract.
<<, >>	Shift left, shift right.
==, !=	Equal, not equal.
<=, <, >=, >	Less than or equal to, less than, greater than or equal to, greater than.
&	Bitwise AND.
^	Bitwise XOR.
|	Bitwise OR.
?:	Conditional operator.

Operands are identifiers defined in either the assembly, the C/C++ program, or in the linker directive.

Examples 31. Lexical Conventions – Operators

```
{(_base_ & 0xFFF) + 0x400000 }
{(_end_ - _start_) * 4 }
```

4.1.6 Comments

Comments can be introduced with the “#” character at any position along the line. All following characters until the end-of-line character are then considered to be comments and are ignored by the linker.

Example 32. Lexical Conventions – Comments

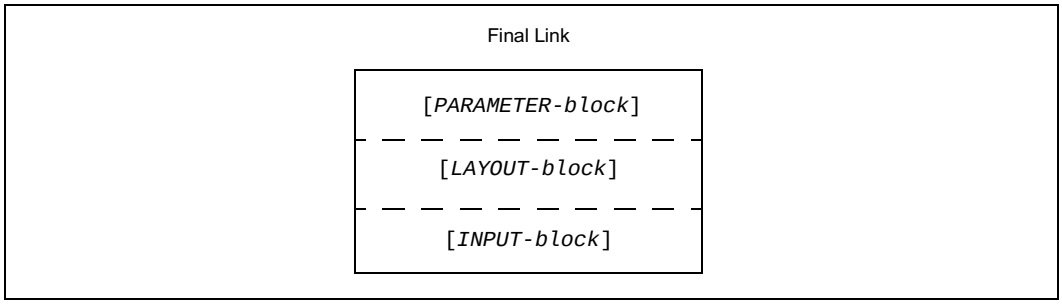
```
# This is a linker build file
# Header block
```

4.2 Build File Syntax

The command line option `-buildfile` enables the use of a build file, which contains additional instructions to the Intel® Linker. A build file contains a sequence of additional commands to the linker in a specific syntax which is detailed below.

A linker build file is built up of several optional blocks, each containing a set of directives. The order of the blocks is fixed and cannot be changed, and are as shown in the following diagram for a final link:

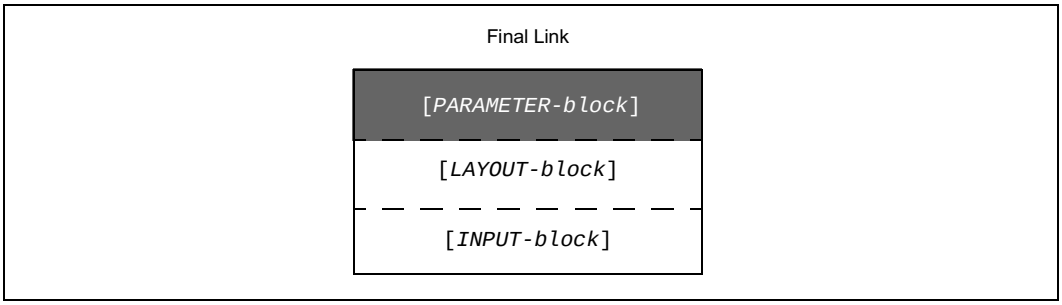
Figure 3. Syntax for a Final Link



The syntax of each block is described in the following sections.

4.2.1 PARAMETER Block

Figure 4. PARAMETER Block




```
PARAMETER-block ::=

    [parameter-option],...
```

where:

```
parameter-option ::=

| ALIGN SECTION  (value | PAGE)

| ALIGN SEGMENT  PAGE value

| ALIGN COMMON  value

| DEFAULT UNDEF (name  |
                 value )

| LABEL (DEFAULT SYMBOLS  |
         SECTION SYMBOLS  |
         MODULES SYMBOLS  |
         SEGMENT SYMBOLS)
```

Description:
The PARAMETER block defines global settings for final link mode, e.g. the allocation of all sections and segments. The PARAMETER block may contain one or several parameter options, for example `ALIGN SECTION`, `ALIGN SEGMENT`, may be specified. They are optional and can be used in any order. These parameters are described in the following sections.

4.2.1.1 ALIGN SECTION

```
ALIGN SECTION  (value | PAGE )
```

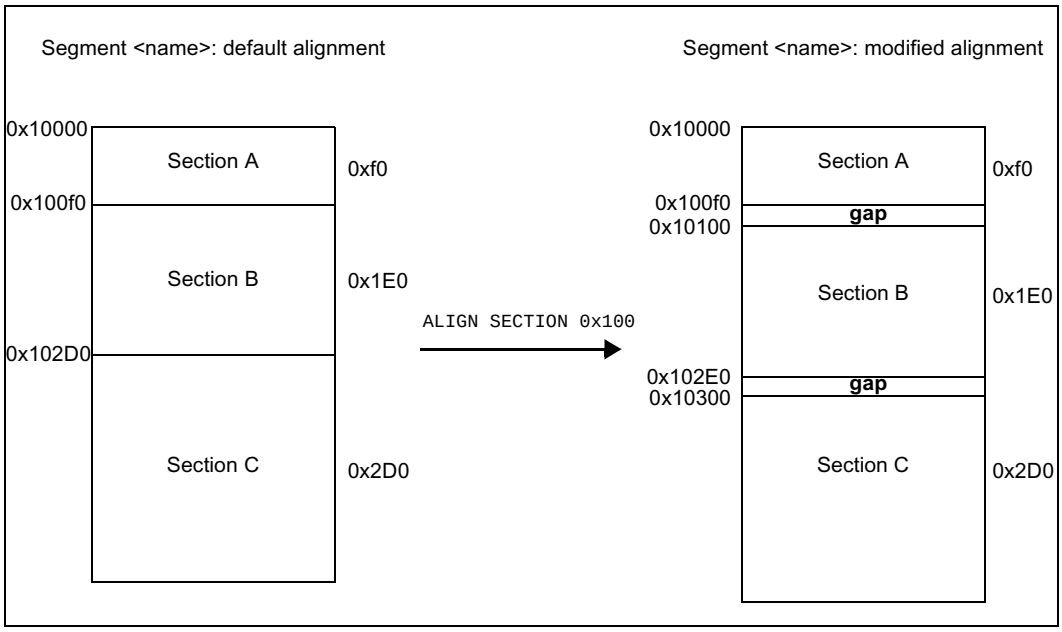
Description:
The `ALIGN SECTION` command specifies the general alignment of all sections, which by default is 4 bytes. Each section may be aligned to a specific edge within the segment. The alignment can be performed as follows:

Option	Description
value	Aligns the start address of sections to the specified value *value*. The second section is aligned at the start address of the first section plus *value*. If the size of the first section is greater than *value*, the second section is aligned at the next possible address which is a multiple of *value*.
PAGE	Aligns the start address of sections to a multiple of 512 bytes which is the value of PAGE. If the size of the first section is greater than PAGE, the next multiple of PAGE is used as start address of the subsequent section.

Example 33. PARAMETER Block – ALIGN SECTION

The following code fragment is used as an example:

```
ALIGN SECTION 0x100
```



The left side of the diagram above shows an example for the default alignment if `ALIGN SECTION` is not used. The right side of the diagram shows the alignment if the above example code is used to align the sections. Since a value of 0x100 is used, a gap appears between section A and B as well as between Section B and C. For data sections the gaps are filled with numeric zeros. For code sections the gaps are filled with NOP instructions.

4.2.1.2 ALIGN SEGMENT

```
ALIGN SEGMENT  (value | PAGE )
```

Description:
The ALIGN SEGMENT command specifies the general alignment of all segments, the default alignment setting of which si 4 bytes.

Option	Description
value	Aligns the start address of segments to the specified value *value*. The second segment is aligned at the start address of the first segment plus *value*. If the size of the first segment is greater than *value*, the second segment is aligned at the next possible address which is a multiple of *value*.
PAGE	Aligns the start address of segments to a multiple of 512 bytes which is the value of PAGE. If the size of the first segment is greater than PAGE, the next multiple of PAGE is used as start address of the subsequent segment.

4.2.1.3 ALIGN COMMON

```
ALIGN COMMON  value
```

Description:
When the ALIGN COMMON command is specified, the Intel® Linker aligns the sizes of all common symbols to the specified value *value*. The default alignment of common symbols is 4 bytes.

4.2.1.4 DEFAULT UNDEF

```
DEFAULT UNDEF (name | value)
```

Description:
The DEFAULT UNDEF directive specifies the allocation of undefined symbols in the output file. The undefined symbol is specified by *name*. This feature enables linking of programs with undefined symbols. You can assign a special address to them to control the behavior of your application, if such a symbol is referenced.

The allocation can be performed as follows:

Option	Description
name	The undefined symbols are located at the address of the symbol *name*. If the mentioned symbol is also undefined, an error message is issued.
value	The undefined symbols are located at the absolute address of *value*.

4.2.1.5 LABEL

```
LABEL ( SECTION  |
        SEGMENT  |
        MODULES  |
         DEFAULT   ) SYMBOLS
```

Description:

The LABEL directive creates global labels, which may be created as follows:

Option	Description	Generated Labels
SECTION SYMBOLS	Creates global labels with an address which is the start and end address of the sections.	_sec_*secname*_beg_ _sec_*secname*_end_ for every section; *secname* is the section name.
SEGMENT SYMBOLS	Creates global labels with an address which is the start and end address of the segments.	_seg_seg*name*_beg_ _seg_*segname*_end_ for every segment; *segname* is the segment name.
MODULES SYMBOLS	Creates global labels with an address which is the start and end address of the sections of input modules.	_*modname*_sec_*secname*_beg_ _*modname*_sec_*secname*_end_ *modname* is the module name and *secname* the section name.
DEFAULT SYMBOLS	Creates global labels with an address which is the start and end address of the segments, sections, and input modules (all of above).	

4.2.1.6 Example for PARAMETER Block

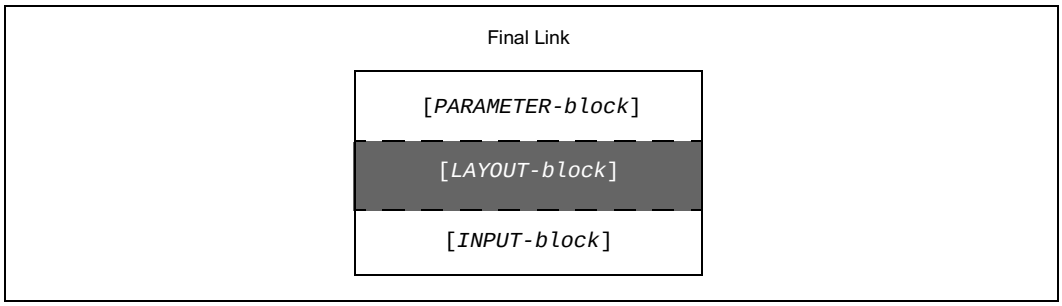
The PARAMETER block of a build file can have the following contents:

```
ALIGN SECTION 0x10
ALIGN SEGMENT 0x10
ALIGN COMMON 0x100
DEFAULT UNDEF ERRORHANDLER
LABEL MODULES
```

All sections are aligned to the value 0x10 or a multiple of 0x10 inside the segment. All segments are aligned to the value 0x10 or a multiple of 0x10. All common symbols are aligned to the value 0x100. The symbol ERRORHANDLER is inserted into the symbol table as undefined. Global labels with an address which is the start and end address of the sections of input modules are created.

4.2.2 LAYOUT Block

Figure 5. LAYOUT Block




```
[LAYOUT-block] ::=

        LAYOUT segment-spec [ segment-spec ],...
```

where:

```
segment-spec ::=

SEGMENT name base-addr
        attr-spec
        layout-body
```

Description:

The LAYOUT block *LAYOUT-block* defines a specific layout of the object file in physical memory. For each segment, specified by *name*, a layout can be defined. The layout for each segment is defined by a base address *base-addr*, attributes *attr-spec*, and a layout body *layout-body*. If the LAYOUT block is not used in the build file, the default layout is assumed. For a description of the default layout, refer to Section 2.6, “Allocation of Segments and Sections” on page 22.

The Intel® Linker accepts segment overlays. Take the example where segment A starts at address 0x1000 and has a size of 0x10000. Then segment B is permitted to start at 0x1100 even though it would be starting within the segment A. In case of an overlay, the Intel® Linker produces a warning message. When a listing file is generated, the “ovlry” column in the memory usage section displays an asterisk for the segment that overlays another segment.

Further Information:

Section 2.6, “Allocation of Segments and Sections” on page 22
Section 3.3, “-base” on page 30

4.2.2.1 Base Address

```
base-addr ::=

  BASE name OF file

| BASE value

| BASE "{" expression "}"

| BASE ALIGN ( value |
                PAGE )
```

Description:

The base address *base-addr* defines the start address of the segment. The base address may be specified as follows:

Base Address	Description
BASE *name* OF *file*	The base address of the segment is the value of the symbol name *name* defined by the object file *file*.
BASE *value*	Sets the base address to the address *value*. *value* is a hexadecimal, decimal, or octal number. No alignments are performed. Odd values cause a warning message.
BASE "{" *expression* "}"	The base address of the segment is set to the value of *expression*.
BASE ALIGN (*value* | PAGE)	This directive is only useful if more than one segment is loaded. BASE ALIGN may have one of the following options:

Option	Description
value	Aligns the start address of segments to the specified value *value*. The second segment is aligned at the start address of the first segment plus *value*. If the size of the first segment is greater than *value*, the second segment is aligned at the next possible address which is a multiple of *value*.
PAGE	Aligns the start address of segments to a multiple of 512 bytes which is the value of PAGE. If the size of the first segment is greater than PAGE, the next multiple of PAGE is used as start address of the subsequent segment.

4.2.2.2 Attributes

```
attr-spec ::=

| QUOTA value

| LOCATION value

| LOCATION "{" expression "}"

| DYNAMIC
```

Description:

The layout of a segment may be defined by using additional attributes as follows:

Option	Description
QUOTA *value*	A maximum segment size of *value* bytes is defined. An error message is issued if a segment exceeds this value. If no quota is defined, the largest unsigned 32-bit value (0xFFFFFFFF) is assumed.
LOCATION *value*	Sets the segment load address to *value*. The base address for re-location is not affected. Therefore, a segment can be relocated to another address during loading. If any absolute section lies inside this segment, a warning is issued. Furthermore, it is useful if segments are stored in ROM at an address which is different to the base address and are copied to the base address after the system start-up. The LOCATION directive can handle global labels as parameter. If the label *name* was defined by a LABEL GLOBAL *name* directive, it can be used in a subsequent directive. See also Section 4.2.1.5, “LABEL” on page 82.
LOCATION "{" *expression* "}"	This option has the same affect as option LOCATION *value* except that the segment load address is set to the address of the specified expression *expression*.
DYNAMIC	Sets the segment as being dynamic, that is it may require the shared resources of other modules or libraries in order to run. See Section 4.2.2.2.1, “Dynamic Segments” on page 87

4.2.2.2.1 Dynamic Segments

The Intel® Linker supports both static and dynamic linking models. In the static model the given object files, system libraries and library archives are statically bound. All global symbol references are resolved and an executable file is created which is completely self contained. While in the dynamic model some of the global symbols are not resolved within the module being linked. The linker converts these into dynamic relocations which will be resolved at load time. The resultant executable is not self contained and at load time other shared resources and dynamic libraries must be made available within the system for the program to run.

The attribute DYNAMIC is used to mark a segment as being dynamic. That is, any unresolved global symbols within such a segment are placed within the special dynamic relocation section of the executable file. Upon loading this executable the loader attempts to resolve all entries within the dynamic relocation section by dynamically linked library modules and other shared resources.

The options `-export` and `-import` are used to control the contents of the dynamic relocation section. Global symbols resolved in this module can be placed into the dynamic relocation table by using the `-export` option. This then makes such global symbols available to be used by other modules, where the symbol is unresolved. The option `-import` may be used to place global symbols into the dynamic relocation section which cannot be resolved within the current module being linked and must therefore be resolved externally. That is, they must be imported from another module, this is carried out at load time.

Another important use of the `-import` option is to force global symbols to be resolved externally, even if they can be resolved internally within other segments of the module being linked. Placing a global symbol into the dynamic relocation section by using the `-import` option, forces the loader to resolve these global symbols externally from the module, and not to use the internally available resolution.

The Intel® Linker supports dynamic linking in both the ARM* and Linux* formats, depending on the option `-dynamic`, which can be set as either `-dynamic=ARM` or `-dynamic=Linux.` The option can also be used without an argument such as `-dynamic` by itself, by default this is taken as being equivalent to `-dynamic=Linux`.

Example 34. Dynamic segment

```
LAYOUT
       SEGMENT .text base 0x0 DYNAMIC    # this is a relocatable DYNAMIC segment
             LOAD     SECTION _RO_

       SEGMENT .data base 0x80000000     # this is an absolute static segment
             LOAD    SECTION _RW_
```

4.2.2.3 Layout Body

```
layout-body ::=

| NOLOAD

  LOAD command ["," command],...
```

where:

```
command ::=

  SECTION section-name

| SECTION section-name OF module ["," module],...

| SECTION section-type

| LABEL (GLOBAL | LOCAL ) name

| DESTRUCTOR

| CONSTRUCTOR
```

Description:

The layout of a segment is defined in detail by the NOLOAD or LOAD command. The NOLOAD command allocates an empty segment. The LOAD command is used to define the allocation of sections within the segment and to apply specific features to the segment.

command may have the following values:

Command	Description
SECTION *section-name*	For every module, the contents of the specified section *section-name* are placed contiguously at the current address counter of this segment. This global assignment of a section is used for every module if there is no module-specific local assignment.
SECTION *section-name* OF *module* ["," *module*],...	For all modules listed in the command, the contents of the specified section *section-name* are placed at the current segment address counter by overriding any global assignment of this section. All specified modules *module* must be specified in the INPUT block. See also Section 4.2.3, "INPUT Block" on page 91.
SECTION *section-type*	A section type can be specified. The section type defines attributes for sections. Available values for *section-type* are explained in the table below.
LABEL GLOBAL *name*	A global label name *name* is defined in the absolute section, and the value of the current address counter is assigned to it. If the label is defined already, an error message is issued.

Command	Description
LABEL LOCAL *name*	A local label name *name* is defined in the absolute section, and the value of the current segment address counter is assigned. It is similar to LABEL GLOBAL *name*, except the label is not written to the final symbol table. If the label is defined already, an error message is issued.
DESTRUCTOR	Loads the destructor table to the specified section within the loaded segment. Only available when option -cplus is set.
CONSTRUCTOR	Loads the constructor table to the specified section within the loaded segment. Only available when option -cplus is set.

Section Type	Description
_RO_	All read-only sections are allocated.
_RW_	All read write sections are allocated.
CODE	All sections containing executable code are allocated.
DATA	All sections containing data are allocated.
ROCODE	All sections that are read-only and contain executable code are allocated.
RWCODE	All sections that are writable and contain executable code are allocated.
RODATA	All sections that are read-only and contain data are allocated.
RWDATA	All sections that are writable and contain data are allocated.
BSS	All sections that contain uninitialized data are allocated.

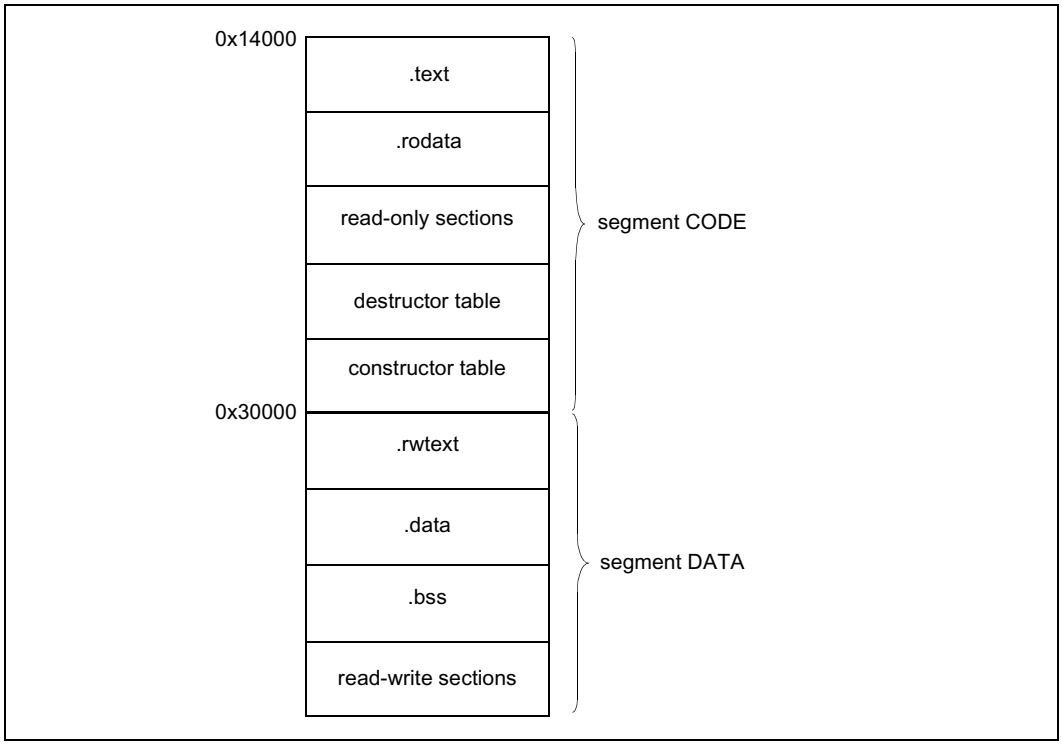
4.2.2.4 Example for LAYOUT Block

The code fragment below is part of a build file:

```
...
...
LAYOUT
    SEGMENT CODE BASE 0x14000
            QUOTA 0x38000
            LOAD SECTION .text
                 SECTION .rodata
                 SECTION _RO_
                 DESTRUCTOR
                 CONSTRUCTOR

    SEGMENT DATA BASE 0x30000
            LOAD SECTION .rwtext
                 LABEL GLOBAL mylabel
                 SECTION .data
                 SECTION .bss
                 SECTION _RW_
...
```

. . .



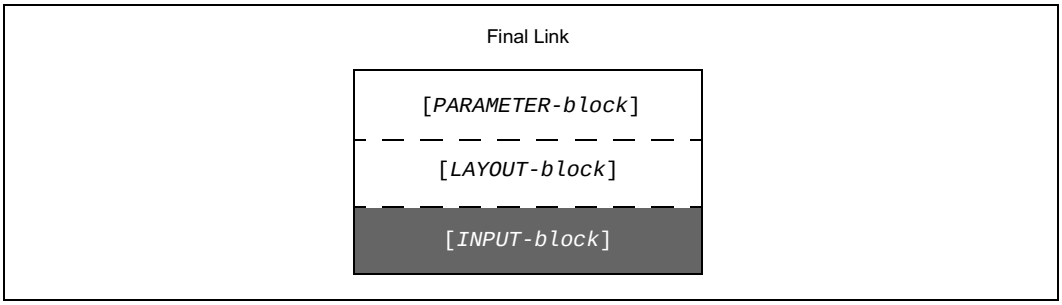
The layout of the output file defines two segments. Segment CODE starts at a base address of 0x14000. The maximum segment size of CODE is set to 0x38000. Segment DATA starts at a base address of 0x30000. Segment CODE contains the sections `.text`, `.rodata`, and read-only sections. The destructor and constructor tables are allocated after all read-only sections inside the segment CODE. Segment DATA contains the sections `.rwtext`, `.data`, `.bss` and read-write sections. Section `.rwtext` starts at the base address of segment DATA followed by the remaining sections. After sections `.rwtext` the global label `mylabel` is defined.

Further Information:

Section 2.6, “Allocation of Segments and Sections” on page 22

4.2.3 INPUT Block

Figure 6. INPUT Block




```
INPUT-block ::=

INPUT input-command ["," input-command],...
```

where:

```
input-command ::=

  module-or-library-name

| library-name "(" module [module],... ")"

| library-name "(" "*" ")"

| _INPUT_

| LABEL (GLOBAL | LOCAL) SECTION section-name name

| LABEL (GLOBAL | LOCAL) "{" expression "}" name

| LABEL (GLOBAL | LOCAL) value name

| LABEL (GLOBAL | LOCAL) name alias

| UNDEF name
```

Description:

The INPUT block *INPUT-block* defines additional input modules and libraries for the Intel® Linker. It is also possible to define re-locatable absolute labels or to create an undefined reference here. The order of commands is important because the commands are processed sequentially.

As input file(s) *input-command* modules and/or libraries may be specified. The file is opened, and its type is checked to determine the appropriate actions. If it is a re-locatable object file, the internal symbol table of the Intel® Linker is updated with the module-specific symbol table. The section sizes are added to the layout description, and the module is inserted into the module chain for further use. If the file is an object library, the library symbol table generated by the Intel® Library Manager is searched for currently undefined symbols. If such a symbol is found, the related module is extracted from the library and added as an ordinary module. The library is searched as long as modules are found. Note, that a library may be searched multiple times.

Option	Description
module-or-library-name	Specifies a module or a library as input file.
library-name "(" *module* [*module*],... ")"	Specifies a library followed by a list of modules. Modules are separated by whitespaces. The specified modules are always extracted from the library, even if they do not define any undefined symbols.
library-name "(" "*" ")"	Specifies a library and all modules in the library; thus, the library is read in completely.
INPUT	Input files can be specified on the command line or in a build file using the directive _INPUT_. The following rule applies: • Objects files specified on the command line are passed to the linker first. Then object files specified inside the build file are passed to the linker. • If the build file contains the directive _INPUT_ inside the build file, the object files are passed to the linker as specified in the build file. Object files specified on the command line are passed to the linker where the directive _INPUT_ is used.
LABEL (GLOBAL | LOCAL) SECTION *section-name name*	A label named *name* is created at section *section-name*. If GLOBAL is specified, a label is placed in the final symbol table in the output file. If LOCAL is selected, the label is available during the link process only.
LABEL (GLOBAL | LOCAL) "{" *expression* "}" *name*	A label named *name* is created at the address of *expression*. If GLOBAL is specified, a label is placed in the final symbol table in the output file. If LOCAL is selected, the label is available during the link process only.

Option	Description
LABEL (GLOBAL | LOCAL) *value name*	A label named *name* is created at the address specified by *value*. *value* is a hexadecimal, decimal, or octal value. If GLOBAL is specified, a label is placed in the final symbol table in the output file. If LOCAL is selected, the label is available during the link process only.
LABEL (GLOBAL | LOCAL) *name alias*	An alias *alias* is created for label named *name*. If GLOBAL is specified, a label is placed in the final symbol table in the output file. If LOCAL is selected, the label is available during the link process only.
UNDEF *name*	An undefined label name *name* is created. This is sometimes useful for forcing the extraction of modules from object libraries. If the symbol is already defined, this command has no effect.

Example 35. INPUT Block

The code below is the fragment of a build file:

```
INPUT
    LABEL GLOBAL root _baseadr_
    entry.o
    main.o
    LABEL GLOBAL SECTION alpha appbeg
    switchtab.o
    lib.lib (calldef.o, userdef.o)
    userlib.lib
    LABEL GLOBAL SECTION beta append
    UNDEF _anchor_
    param.lib
    end.o
```

The build file defines the following:

- The global label `_baseadr_` is created at the same address as `root`. The object modules `entry.o` and `main.o` are read in.
- The global label `appbeg` is created at the address of section `alpha`.
- The module `switchtab.o` is read in.
- The modules `calldef.o` and `userdef.o` are extracted from the library `lib.lib`.
- The library `userlib.lib` is searched for any undefined symbols.
- The global label `append` is created at the address of section `beta`.
- The symbol `_anchor_` is inserted into the symbol table as undefined.
- The library `param.lib` is processed followed by the module `end.o`.

Example 36. INPUT Block – Reading Objects

Assuming the contents of the build file is as follows:

```
INPUT
   x.o
   _INPUT_
   y.o
   z.o

ldxsc -bf b.o a.o
```

The command line above processes the object files in the following order:

1. x.o
2. b.o
3. a.o
4. y.o
5. z.o

Further Information:
Section 3.4, “-buildfile” on page 31

Listing File 5

This chapter describes the listing file that may be created during the link process. Such a listing file contains helpful information that can be used to detect errors of the link process.

The chapter contains the following sections:

- Section 5.1, “Creating a Listing File” on page 96 provides an overview how you can create a listing file using specific command line options.
- Section 5.2, “Contents of a Listing File” on page 98 provides an overview of the contents of a listing file and describes the listing file in detail.

5.1 Creating a Listing File

A listing file provides helpful information of the entire link process. It is created using either the option `-listing` or `-listing=`*`name`*. Both options create a listing file containing all information as described in Section 5.2. When the option `-info` is specified in addition to `-listing` or `-listing=`*`name`*, a listing file is created containing specific information. The filename of the listing file depends on the used options. See Table 7.

Table 7. Listing File – Options and Filenames

Option	Description		Filename and File Extension
`-listing`	Creates a listing file.		The name of the listing file consists of the base name of the first input object and the extension .map. If an output file is specified with the option `-output`, the name of the listing file is the base name of the output file with extension .map.
`-listing=`*`name`*	Creates a listing file with name *`name`*.		If no extension is specified, the default extension of the listing file is “map”.

`-listing`
`-info=`*`arg`*`{,`*`arg`*`}`
 	
or
 	
`-listing=`*`name`*
`-info=`*`arg`*`{,`*`arg`*`}`
	Specifies parameters for the listing file. Available values for *`arg`* are:		Depends upon `-listing` or `-listing=`*`name`* being used. See above.

	`buildfile`	Prints the contents of the build file in the listing file. See also Section 5.2.10, “Build File” on page 109.	
	`errors`	Prints diagnostic messages inside the header in the listing file. See also Section 5.2.1, “Header” on page 99.	
	`map`	Prints the symbol table in the listing file. See also Section 5.2.8, “Symbol Table” on page 106.	
	`memory`	Prints the memory usage in the listing file. See also Section 5.2.5, “Memory Usage” on page 103.	
	`modules`	Prints the module summary in the listing file. See also Section 5.2.4, “Module Summary” on page 102.	
	`sections`	Prints the segment-section summary in the listing file. See also Section 5.2.3, “Segment-Section Summary” on page 101.	
	`segments`	Prints the segment summary in the listing file. See also Section 5.2.2, “Segment Summary” on page 100.	
	`symbols`	Prints the listing of module symbols and segment symbols in the listing file. See also Section 5.2.6, “Listing of Module Symbols” on page 104 and Section 5.2.7, “Listing of Segment Symbols” on page 105.	

Examples 37. Listing File – File Extension

```
ldxsc -listing -info=errors a.o b.o
```

The generated listing file is **a.map**.

```
ldxsc -listing -info=errors -o test a.o
```

The generated listing file is **test.map**.

```
ldxsc -listing=list a.o
```

The generated listing file is **list.map**.

```
ldxsc -listing=list.lst a.o
```

The generated listing file is **list.lst**.

Further Information:

Section 3.20, “-info” on page 53
Section 3.23, “-listing” on page 57
Section 5.2, “Contents of a Listing File” on page 98

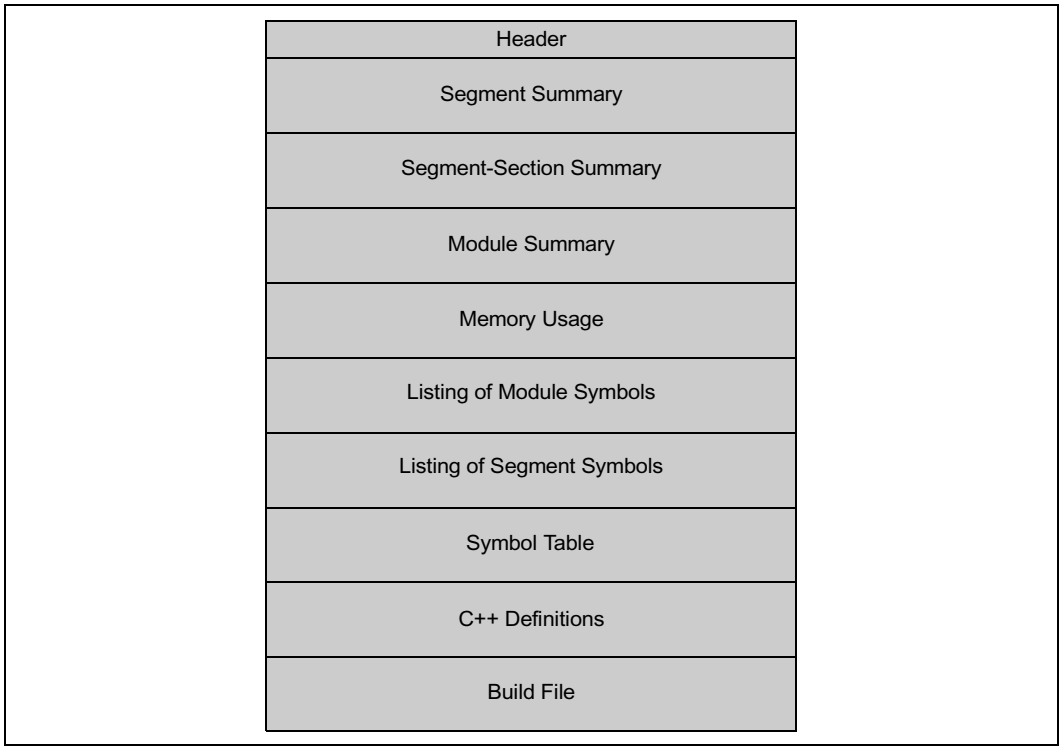
5.2 Contents of a Listing File

The listing file is a simple ASCII file that contains helpful information on the link process, symbols, modules, C++ definitions, and the used build file. Figure 7 shows the contents of a listing file. Information is placed subsequently, separated by a line "= = = = = =". The listing file starts with a header, followed by a summary of diagnostic messages, summaries of segments, sections, and modules. Then the memory usage, listing of module and segment symbols and the symbol table are displayed. Following this, provided the option `-cplus` is set, is a list of C++ definitions. Finally the list file ends with the contents of any used build file.

The end address of sections, segments, and memory blocks is calculated as follows:

```
endaddress - startaddress = size
```

Figure 7. Listing File



5.2.1 Header

The header of a listing file provides general information on the tool and its version, the link time, the used command line, and diagnostic messages. The header is printed in every listing file by default.

The diagnostic messages are placed in the listing file by default unless the option `-info` is specified. In which case they are only listed if the option is specified as `-info=errors`. Setting the option `-nowarnings` suppresses the warning messages in the listing.

If the option `-verbose` is set, the additional verbose output is also written to the listing file.

Example 38. Listing File – Header

```
Intel(R) Linker for Intel(R) XScale(TM) Microarchitecture, Version: version


Link Time: Tue Apr 17 14:51:03 2001


Command Line:

ldxsc -listing -o out b.o

LDXSC-W-WARNING[666]:following symbols are undefined:
LDXSC-W-WARNING[667]:b.o:symbol : __code_start__ weak
LDXSC-W-WARNING[667]:caller.o:symbol foo: weak

      3 warnings
```

Table 8. Listing File – Header

Item	Description
first line	Displays the tool name and its version.
`Link Time`	Displays the time and date of the link process.
`Command Line`	Displays the command line, including executable, build file, and command line options.

5.2.2 Segment Summary

The segment summary provides information on every segment used in the entire application. The listing file contains base and end address, size, type, and attributes of all segments. This segment summary is listed by default in the listing file unless the option `-info` is specified. In which case no segment summary will be listed unless the option is specified as `-info=segments`.

Example 39. Listing File – Segment Summary

```
Segment Summary:

Base Addr   Total       End Addr     Type  Attr  Segment

0x00008000  0x00002060  0x0000a060   Code  RO    CODE
0x00000000  0x0000a16c  0x0000a16c   Zero  RW    BSS
```

Table 9. Listing File – Segment Summary

Item	Description
`Base Addr`	Displays the base address of the segment as hexadecimal value.
`Total`	Displays the size of the segment as hexadecimal value.
`End Addr`	Displays the end address of the segment as hexadecimal value.
`Type`	Displays the segment type. This is either Code, Data, or Zero.
`Attr`	Displays the attribute of the segment. The segment is either read-only (RO) or read-write (RW).
`Segment`	Displays the segment name.

5.2.3 Segment-Section Summary

The segment-section summary of a listing file lists information on every section contained in a segment. The information is displayed for each segment, separated by a line "--------". This segment-section summary is listed by default unless the option `-info` is specified. In which case no segment-section summary will be listed unless the option is specified as `-info=sections`.

Example 40. Listing File – Segment-Section Summary

```
Segment - Section Summary:

Segment: CODE
Base Addr   Total       End Addr     Type  Attr  Section
0x00008000  0x00001efc  0x00009efc   Code  RO    *
0x00009efc  0x00000144  0x0000a040   Data  RO    *
0x0000a040  0x00000004  0x0000a044   Data  RO
0x0000a044  0x0000001c  0x0000a060   Data  RO
--------------------------------------------------------------------------------
Segment: BSS
Base Addr   Total       End Addr     Type  Attr  Section
0x0000a060  0x0000010c  0x0000a16c   Zero  RW    *
--------------------------------------------------------------------------------
```

Table 10. Listing File – Segment-Section Summary

Item	Description
`Segment`	Displays the segment name.
`Base Addr`	Displays the base address of the section.
`Total`	Displays the size of the section.
`End Addr`	Displays the end address of the section.
`Type`	Displays the section type. This is either Code, Data, or Zero.
`Attr`	Displays the attribute of the section. The section is either read-only (RO) or read-write (RW).
`Section`	Displays the section name. An asterisk "*" means that the section has any name. If no name is specified, the section was generated by the linker.

5.2.4 Module Summary

The module summary displays information on every module contained within the application or in a library. If a module belongs to a library, the library name is displayed in parentheses "`()`". This module summary is listed by default unless the option `-info` is specified. In which case no module summary will be listed unless the option is specified as `-info=modules`.

Example 41. Listing File – Module Summary

```
Module Summary:

Module(Library)

main.o

    Section                   Base        Size        Align  Segment
    .text                     0x000080a8  0x00000048      4  CODE
    .text$Veneer              0x000080f0  0x00000010      4  CODE

sub.o

    Section                   Base        Size        Align  Segment
    .text                     0x00008100  0x0000002c      4  CODE

printf.o(x0__ac00.a)

    Section                   Base        Size        Align  Segment
    .text                     0x00008130  0x00000038      4  CODE
...
...
...
```

Table 11. Listing File – Module Summary

Item	Description
`Section`	Displays the section name.
`Base`	Displays the base address of the module.
`Size`	Displays the size of the module.
`Align`	Displays the alignment of sections in bytes.
`Segment`	Displays the segment the module belongs to.

5.2.5 Memory Usage

The memory usage of a listing file provides detailed information on each segment. It lists the location of segments parts within the memory in detail, including, start and end address, corresponding section, type and attribute of the segment. This memory usage summary is listed by default unless the option `-info` is specified. In which case no memory usage summary will be listed unless the option is specified as `-info=memory`.

Example 42. Listing File – Memory Usage

```
Memory Usage:

Start       End         Section           Segment       Ovrly  Type  Attr

0x00008000  0x00009efc  ROCODE            CODE                 Code  RO
0x00009000  0x0000a040  RODATA            CODE            *    Code  RO
0x0000a040  0x0000a044  __constructor_se  CODE                 Code  RO
0x0000a044  0x0000a060  __destructor_sec  CODE                 Code  RO
0x0000a060  0x0000a16c  BSS               BSS                  Zero  RW
```

Table 12. Listing File – Memory Usage

Item	Description
`Start`	Displays the start address of the segment.
`End`	Displays the end address of the segment.
`Section`	Displays the section name.
`Segment`	Displays the segment name.
`Ovrly`	Displays an asterisk in case of a segment overlay. In the example above, section RODATA of segment CODE overlays section ROCODE of segment CODE. Therefore an asterisk appears for section RODATA of segment CODE.
`Type`	Displays the type of the segment. This is either Code, Data, or Zero.
`Attr`	Displays the attribute of the segment. The segment is either read-only (RO) or read-write (RW).

5.2.6 Listing of Module Symbols

The listing of module symbols displays all symbols contained in your application or libraries. If a symbol belongs to a library, the library name is displayed in parentheses "`( )`". The module symbols are listed by default unless the option `-info` is specified. In which case no symbols will be listed unless the option is specified as `-info=symbols`.

Example 43. Listing File – Listing of Module Symbols

```
Listing of Module Symbols:

Section           Address     Size        Type  Symbol

main.o

<ref>                                             Lib$$Request$$lib
<ref>             0x00008000  0x00000098        __main
<ref>             0x0000812e  0x00000002        _main
<ref>             0x00008130  0x0000002c        _printf
.text             0x000080a8  0x00000048  Code  main
<ref>             0x00008100  0x0000000a        function

sub.o

<ref>                                             Lib$$Request$$lib
<ref>             0x00008130  0x0000002c        _printf
.text             0x00008100  0x0000000a  Code  function

printf.o(x0__ac00.a)

<ref>                                             Lib$$Request$$lib
<ref>             0x0000a0e4  0x00000044        __stdout
<ref>             0x00008bf4  0x000005be        __vfprintf
.text             0x00008130  0x0000002c  Code  _printf
<ref>             0x00008168  0x00000008        ferror
<ref>             0x00008170  0x00000024        fputc
...
```

Table 13. Listing File – Listing of Module Symbols

Item	Description
`Section`	Displays the section name the module symbol belongs to. If a section name `<ref>` is displayed, the symbol is referenced within the section only. If a section name is specified, the symbol is defined within this section.
`Address`	Displays the start address the symbol is placed within the module.
`Size`	Displays the size of the symbol.
`Type`	Displays the type of the section. This is either Code, Data, or Zero.
`Symbol`	Displays the symbol name.

5.2.7 Listing of Segment Symbols

The listing of segment symbols lists symbols that are defined in the build file or on the command line by the user. These segment symbols are listed by default unless the option `-info` is specified. In which case no segment symbols will be listed unless the option is specified as `-info=symbols`.

Example 44. Listing File – Listing of Segment Symbols

```
Listing of Segment Symbols:

Address     Scope     Symbol

Segment CODE:

0x00008000  Global    __code_start__
0x0000804c  Global    __code_end__
Segment DATA:

0x00009000  Global    __data_start__
0x00009004  Global    __data_end__
Segment BSS:

0x0000a000  Global    __bss_start__
0x0000a028  Global    __bss_end__
```

Table 14. Listing File – Listing of Segment Symbols

Item	Description
`Address`	Displays the absolute address of the symbol in the memory.
`Scope`	Displays the scope of the symbol. This is either global or local. See also Section 4.2.1.5, “LABEL” on page 82 and Section 4.2.2.3, “Layout Body” on page 88.
`Symbol`	Displays the symbol name.

5.2.8 Symbol Table

The symbol table lists the entry point of the symbol table as hexadecimal value, displays mapping symbols and global symbols. The symbol table is listed by default unless the option `-info` is specified. In which case symbol table is listed unless the option is specified as `-info=map`.

Example 45. Listing File – Symbol Table

```
Symbol Table:

Entry Point: 0x00008000

--------------------------------------------------------------------------------

Mapping Symbols:

Segment       Address     Symbol

CODE          0x00008000  $a
CODE          0x00008098  $d
CODE          0x000080a8  $a
CODE          0x000080c8  $d
CODE          0x00008100  $t
CODE          0x00008104  $b
CODE          0x0000810a  $d
CODE          0x0000812c  $t
CODE          0x00008160  $f
CODE          0x00008168  $t

--------------------------------------------------------------------------------

Global Symbols:

Segment    Section           Address     Size        Type  Symbol

ABSOLUTE   <abs>             0x00000000                    Image$$RW$$Limit
CODE       <abs>             0x00009a10                   _rtudiv10_sec_.text_beg_
CODE       .text             0x00009a14                Code  __rt_udiv10
CODE       <abs>             0x00009a40                   _rtudiv10_sec_.text_end_
CODE       <abs>             0x00009a40                    _setvbuf_sec_.text_beg_
CODE       .text             0x00009a40  0x00000056  Code  setvbuf
CODE       .text             0x00009a96  0x0000001a  Code  setbuf
CODE       <abs>             0x00009ab4                    _setvbuf_sec_.text_end_
CODE       <abs>             0x00009ab4                    _strlen_sec_.text_beg_
CODE       .text             0x00009ab4  0x00000046  Code  strlen
```

Table 15. Listing File – Symbol Table

Item	Description
`Entry Point`	Displays the entry point of the symbol table.
Mapping Symbols	
`Segment`	Displays the segment name the symbol belongs to.
`Address`	Displays the address the symbol is placed within the segment.
`Symbol`	Displays the name of the Intel XScale microarchitecture mapping symbol. These symbols, which are generated by the compiler or assembler, indicate changes between opcode modes. See also Table 16.
Global Symbols	
`Segment`	Displays the segment name the symbol belongs to. An ABSOLUTE segment indicates that the symbol is generated by the linker and does not belong to any segment.
`Section`	Displays the section name the symbol belongs to.
`Address`	Displays the absolute address where the symbol is placed.
`Size`	Displays the size of the symbol. If no size is specified, it is not known by the linker.
`Type`	Displays the type of the section. This is either Code, Data, or Zero.
`Symbol`	Displays the symbol name.

Table 16. IIntel XScale Microarchitecture Mapping Symbols

Symbol	Description
$a	Indicates the change from Thumb* code to ARM* code.
$b	Indicates Thumb* BL instruction.
$d	Indicates data.
$f	Points to a function pointer.
$t	Indicates the change from ARM* code to Thumb* code.

5.2.9 C++ Definitions

The C++ definition part of a listing file displays information on C++-specific symbols and table names. This part of the listing file is displayed only if the option `-cplus` is set.

```
C++ DEFINITIONS:

    CONSTRUCTOR SYMBOLS: __sti* (count=0)

    DESTRUCTOR  SYMBOLS: __std* (count=6)

Section           Address     Size        Type  Symbol

.text             0x0000a128  0x00000044        __stderr
.text             0x00009f04  0x00000004        __stderr_name
.text             0x0000a0a0  0x00000044        __stdin
.text             0x00009efc  0x00000004        __stdin_name
.text             0x0000a0e4  0x00000044        __stdout
.text             0x00009f00  0x00000004        __stdout_name
-------------------------------------------------------------------------------


    CONSTRUCTOR TABLE  : __ctors at 0x0000a040

    DESTRUCTOR  TABLE  : __dtors at 0x0000a044
```

Table 17. Listing File – C++ Definitions

Item	Description
`CONSTRUCTOR SYMBOLS`	Displays the name of the constructor symbol.
`DESTRUCTOR SYMBOLS`	Displays the name of the destructor symbol.
`Section`	Displays the section name the symbol belongs to. If the section name `<ref>` is specified, the symbol is referenced in the section only.
`Address`	Displays the absolute address of the symbol.
`Size`	Displays the size of the symbol.
`Type`	Displays the type of the section. This is either Code, Data, or Zero.
`Symbol`	Displays the symbol name.
`CONSTRUCTOR TABLE`	Displays the address where the constructor table is placed in memory.
`DESTRUCTOR TABLE`	Displays the address where the destructor table is placed in memory.

5.2.10 Build File

The build file part of a listing file displays the entire contents of the build file that is used for the link process. The build file is listed in the listing file only if the option `-info=buildfile` is set.

```
Build File

! Input: buildfile.bld

layout
    default base 0x8000
input
    main.o
    sub.o
    label global 0 Image$$RW$$Limit
    label global 0 Region$$Table$$Base
    label global 0 Region$$Table$$Limit
```

This page intentionally left blank.

Diagnostic Messages 6

When problems are encountered the Intel® Linker generates suitable messages. These include errors, warnings and diagnostic messages which are generated in differing formats.This chapter describes these message formats.

6.1 Message Format

The diagnostic message format is as follows:

`LDXSC`*`qualifier`*`:[`*`messagenumber`*`]:`*`file`*`:`*`message`*

where:

qualifier	Specifies the type of diagnostic message: empty -W-WARNING -E-ERROR -E-FATAL
messagenumber	Specifies the message number.
file	Specifies the filename the message relates to.
message	Is a short description of the problem.

The following table lists the qualifiers with a short description:

Table 18. Characterization of Diagnostic Messages

Qualifier	Description
empty	Specifies an information message. It informs about details in the ongoing process that are neither problems nor errors.
-W-WARNING	Specifies a warning message. This message informs the user about a problem within the application.
-E-ERROR	Specifies a non-fatal error message. An error during the execution of the Intel® Linker has been reported. This problem must be solved.
-E-FATAL	Specifies a fatal error message. The reported error is more significant and has prevented the Intel® Linker from continuing its work. This problem must be solved before anything else can be done using the Intel® Linker.

Appendix – Reserved Words A

Table 19 contains all reserved words. If you use one of these words as a symbol, for example a variable name, inside a build file, it must be 'escaped' by preceeding it with the ^ character, as described in Section 4.1.2, "Keywords" on page 74.

Table 19. Reserved Words

BSS	CONSTRUCTOR	MODULE
CODE	DEFAULT	NOLOAD
DATA	DESTRUCTOR	OF
INPUT	DYNAMIC	PAGE
_RO_	END	QUOTA
ROCODE	GLOBAL	REGISTER
RODATA	INPUT	RELOCATABLE
_RW_	LABEL	SECTION
RWCODE	LABFILE	SEGMENT
RWDATA	LAYOUT	SELECTOR
ALIGN	LOAD	SPACE
BASE	LOCAL	SYMBOLS
COMMON	LOCATION	UNDEFINED

This page intentionally left blank.

Index

Symbols

A

B

C

D

E

F

M

N

O

This page intentionally left blank.