Smart Linker

Revised 04/29/2003

Metrowerks, the Metrowerks logo, and CodeWarrior are registered trademarks of Metrowerks Corp. in the US and/or other countries. All other tradenames and trademarks are the property of their respective owners.

Copyright © Metrowerks Corporation. 2003. ALL RIGHTS RESERVED.

The reproduction and use of this document and related materials are governed by a license agreement media, it may be printed for non-commercial personal use only, in accordance with the license agreement related to the product associated with the documentation. Consult that license agreement before use or reproduction of any portion of this document. If you do not have a copy of the license agreement, contact your Metrowerks representative or call 800-377-5416 (if outside the US call +1-512-996-5300). Subject to the foregoing non-commercial personal use, no portion of this documentation may be reproduced or transmitted in any form or by any means, electronic or mechanical, without prior written permission from Metrowerks.

Metrowerks reserves the right to make changes to any product described or referred to in this document without further notice. Metrowerks makes no warranty, representation or guarantee regarding the merchantability or fitness of its products for any particular purpose, nor does Metrowerks assume any liability arising out of the application or use of any product described herein and specifically disclaims any and all liability. **Metrowerks software is not authorized for and has not been designed, tested, manufactured, or intended for use in developing applications where the failure, malfunction, or any inaccuracy of the application carries a risk of death, serious bodily injury, or damage to tangible property, including, but not limited to, use in factory control systems, medical devices or facilities, nuclear facilities, aircraft navigation or communication, emergency systems, or other applications with a similar degree of potential hazard**.

USE OF ALL SOFTWARE, DOCUMENTATION AND RELATED MATERIALS ARE SUBJECT TO THE METROWERKS END USER LICENSE AGREEMENT FOR SUCH PRODUCT.

How to Contact Metrowerks

Corporate Headquarters	Metrowerks Corporation 7700 West Parmer Lane Austin, TX 78729 U.S.A.
World Wide Web	`http://www.metrowerks.com`
Sales	Voice: 800-377-5416 Fax: 512-996-4910 Email: sales@metrowerks.com
Technical Support	Voice: 800-377-5416 Email: support@metrowerks.com

Table of Contents

1 Introduction 15

Notation . . . 15
Structure of this Document . . . 15
Purpose of a Linker . . . 16

2 User Interface 17

Tip of The Day Dialog . . . 17
Main Window . . . 18
Window Title . . . 18
Content Area . . . 19
Tool Bar . . . 20
Status Bar . . . 21
Linker Menu Bar . . . 21
Options Settings Dialog Box . . . 30
Message Settings Dialog Box . . . 31
About Box . . . 33
Retrieving Information about an Error Message . . . 34
Specifying the Input File . . . 35
Message/Error Feedback . . . 36

3 Environment 39

The Current Directory . . . 40
Global Initialization File (MCUTOOLS.INI) (PC only) . . . 40
[Installation] Section . . . 41
[Options] Section . . . 41
[LINKER] Section . . . 42
[Editor] Section . . . 43
Example . . . 44
Local Configuration File (usually project.ini) . . . 45
[Editor] Section . . . 46
[[LINKER] Section . . . 47

Example . . . 50
Paths . . . 51
Line Continuation . . . 51
Environment Variable Details . . . 52
.ABSPATH . . . 53
ABSPATH: Absolute Path . . . 53
COPYRIGHT . . . 54
COPYRIGHT: Copyright Entry in Absolute File . . . 54
DEFAULTDIR . . . 54
DEFAULTDIR: Default Current Directory . . . 54
ENVIRONMENT . . . 55
ENVIRONMENT: Environment File Specification . . . 55
ERRORFILE . . . 56
ERRORFILE: Error File Name Specification . . . 56
GENPATH . . . 58
GENPATH: Define Paths to search for input Files . . . 58
INCLUDETIME . . . 59
INCLUDETIME: Creation Time in Object File . . . 59
LINKOPTIONS . . . 60
LINKOPTIONS: Default SmartLinker Options . . . 60
OBJPATH . . . 60
OBJPATH: Object File Path . . . 60
RESETVECTOR . . . 61
RESETVECTOR: Reset Vector Location . . . 61
SRECORD . . . 61
SRECORD: S Record File Format . . . 61
TEXTPATH . . . 62
TEXTPATH: Text Path . . . 62
TMP . . . 63
TMP: Temporary directory . . . 63
USERNAME . . . 63
USERNAME: User Name in Object File . . . 63

4 Files **65**

Input Files 65
Parameter File 65
Object File 65
Output Files 65
Absolute Files 65
Motorola S Files 66
Map Files 66
Error Listing File 67

5 SmartLinker Options **71**

SmartLinker Option Details 71
-Add 72
-Add: Additional Object/Library File 72
-AllocFirst,-AllocNext,-AllocChange 73
-Alloc: Allocation over segment boundaries (ELF) 73
-AsROMLib 75
-AsROMLib: Link as ROM Library 75
-B 76
-B: Generate S-Record file 76
-CAllocUnusedOverlap 76
-CAllocUnusedOverlap: Allocate not referenced overlap variables (HIWARE) . 76
-Ci 77
-Ci: Link Case Insensitive 77
-Cocc 78
-Cocc: Optimize Common Code (ELF) 78
-CRam 79
-CRam: Allocate non specified const segments in RAM (ELF) 79
-Dist 79
-Dist: Enable distribution optimization (ELF) 79
-DistFile 80
-DistFile: Specify distribution file name (ELF) 80
-DistInfo 80

-DistInfo: Generate distribution information file (ELF) 80
-DistOpti 81
-DistOpti: Choose optimizing method (ELF) 81
-DistSeg. 81
-DistSeg: Specify distribution segment name (ELF) 81
-E 82
-E: Define Application Entry Point (ELF) 82
-Env 82
-Env: Set Environment Variable 82
-FA, -FE, -FH -F6 83
-FA, -FE, -FH -F6: Object File Format 83
-H 84
-H: Prints the List of All Available Options. 84
-L 84
-L: Add a path to the search path (ELF) 84
-Lic 85
-Lic: Print license information 85
-LicA. 85
-LicA: License Information about every Feature in Directory. 85
-M 86
-M: Generate Map File. 86
-N 86
-N: Display Notify Box 86
-NoBeep 87
-NoBeep: No Beep in Case of an Error 87
-NoEnv 87
-NoEnv: Do not use Environment 87
-OCopy 88
-OCopy: Optimize Copy Down (ELF). 88
-O 89
-O: Define Absolute File Name 89
-Prod 89
-Prod: specify project file at startup (PC). 89

-S 90
-S: Do not generate DWARF Information (ELF) 90
-SFixups 91
-SFixups: Creating Fixups (ELF) 91
-StatF 91
-StatF: Specify the name of statistic file 91
-V 92
-V: Prints the SmartLinker Version 92
-View 92
-View: Application Standard Occurrence (PC) 92
-W1 93
-W1: No Information Messages 93
-W2 94
-W2: No Information and Warning Messages 94
-WErrFile 95
-WErrFile: Create "err.log" Error File 95
-Wmsg8x3 95
-Wmsg8x3: Cut file names in Microsoft format to 8.3 (PC) 95
-WmsgCE 96
-WmsgCE: RGB color for error messages 96
-WmsgCF 97
-WmsgCF: RGB color for fatal messages 97
-WmsgCI 97
-WmsgCI: RGB color for information messages 97
-WmsgCU 98
-WmsgCU: RGB color for user messages 98
-WmsgCW 98
-WmsgCW: RGB color for warning messages 98
-WmsgFb (-WmsgFbv, -WmsgFbm) 99
-WmsgFb: Set message file format for batch mode 99
-WmsgFi (-WmsgFiv, -WmsgFim) 100
-WmsgFi: Set message file format for Interactive mode 100
-WmsgFob 101

-WmsgFob: Message format for Batch Mode 101
-WmsgFoi 103
-WmsgFoi: Message Format for Interactive Mode 103
-WmsgFonf 104
-WmsgFonf: Message Format for no File Information 104
-WmsgFonp 106
-WmsgFonp: Message Format for no Position Information 106
-WmsgNe 107
-WmsgNe: Number of Error Messages 107
-WmsgNi 108
-WmsgNi: Number of Information Messages 108
-WmsgNu 108
-WmsgNu: Disable User Messages 108
-WmsgNw 109
-WmsgNw: Number of Warning Messages 109
-WmsgSd 110
-WmsgSd: Setting a Message to Disable 110
-WmsgSe 110
-WmsgSe: Setting a Message to Error 110
-WmsgSi 111
-WmsgSi: Setting a Message to Information 111
-WmsgSw 111
-WmsgSw: Setting a Message to Warning 111
-WOutFile 112
-WOutFile: Create Error Listing File 112
-WStdout 112
-WStdout: Write to standard output 112

6 Linking Issues 115

Object Allocation 115
The SEGMENTS Block (ELF) 115
The SECTIONS Block (HIWARE + ELF) 121
PLACEMENT Block 124

Initializing Vector Table 129
VECTOR Command 129
Smart Linking (ELF) 130
Mandatory Linking from an Object 130
Mandatory Linking from all Objects defined in a File 131
Switching OFF Smart Linking for the Application 131
Smart Linking (HIWARE + ELF). 131
Mandatory Linking from an Object 132
Mandatory Linking from all Objects defined in a File 132
Binary Files building an Application (ELF) 133
NAMES Block 133
ENTRIES Block 133
Binary Files building an Application (HIWARE). 134
NAMES Block 134
Allocating Variables in "OVERLAYS". 135
Example: 135
Overlapping Locals 136
Example: 136
Algorithm 137
Name Mangling for Overlapping Locals 139
Name Mangling in the ELF Object File Format 140
Defining an function with overlapping parameters in Assembler 141
DEPENDENCY TREE in the Map File 146
Optimizing the overlap size 147
Recursion Checks. 147
Linker Defined Objects. 149
Automatic Distribution of Paged Functions 151
Limitations 156
Checksum Computation 156
Prm file controlled Checksum Computation 157
Automatic Linker controlled Checksum Computation 158
Partial Fields. 160
Runtime support 160

Linking an Assembly Application 160
Prm File 160
WARNINGS 161
Smart Linking 161
LINK_INFO(ELF) 164

7 The Parameter File 165

The Syntax of the Parameter File 165
Mandatory SmartLinker Commands. 168
The INCLUDE directive 169

8 SmartLinker Commands 171

AUTO_LOAD 171
AUTO_LOAD: Load Imported Modules (HIWARE, M2) 171
CHECKSUM 172
CHECKSUM: Checksum computation (ELF). 172
CHECKKEYS 176
CHECKKEYS: Check Module Keys (HIWARE, M2) 176
DATA 176
DATA: Specify the RAM Start (HIWARE) 176
DEPENDENCY 177
DEPENDENCY: Dependency Control 177
ROOT 178
USES 178
ADDUSE 179
DELUSE 180
Overlapping of local variables and parameters 181
ENTRIES 182
ENTRIES: List of Objects to Link with the Application. 182
HAS_BANKED_DATA 184
HAS_BANKED_DATA: Application has banked data (HIWARE) 184
HEXFILE 185
HEXFILE: Link a Hex File with the Application 185

INIT . . . 186
INIT: Specify the Application Init Point . . . 186
LINK . . . 187
LINK: Specify Name of Output File . . . 187
MAIN . . . 188
MAIN: Name of the Application Root Function . . . 188
MAPFILE . . . 189
MAPFILE: Configure Map File Content . . . 189
NAMES . . . 192
NAMES: List the Files building the Application . . . 192
OVERLAP_GROUP . . . 194
OVERLAP_GROUP: Application uses Overlapping (ELF) . . . 194
PLACEMENT . . . 196
PLACEMENT: Place Sections into Segments . . . 196
PRESTART . . . 198
PRESTART: Application Prestart Code (HIWARE) . . . 198
SECTIONS . . . 199
SECTIONS: Define Memory Map . . . 199
SEGMENTS . . . 202
SEGMENTS: Define Memory Map (ELF) . . . 202
STACKSIZE . . . 208
STACKSIZE: Define Stack Size . . . 208
STACKTOP . . . 210
STACKTOP: Define Stack Pointer Initial Value . . . 210
START . . . 211
START: Specify the ROM Start (HIWARE) . . . 211
VECTOR . . . 212
VECTOR: Initialize Vector Table . . . 212

9 Sections (ELF) 215

Terms: Segments and Sections . . . 215
Definition of Section . . . 215
Predefined Sections . . . 216

10 Segments (HIWARE) 219

Terms: Segments and Sections (HIWARE) 219
Definition of Segment (HIWARE) 219
Predefined Segments 220

11 Examples 223

Example 1 223
Example 2 223

12 Program Startup 225

The Startup Descriptor (ELF) 225
User Defined Startup Structure: (ELF) 228
Example 229
User Defined Startup Routines (ELF) 230
The Startup Descriptor (HIWARE) 230
User Defined Startup Routines (HIWARE) 232
Example of Startup Code in ANSI-C 232

13 The Map File 239

Map File Contents 239

14 ROM Libraries 241

Creating a ROM Library 241
ROM Libraries and Overlapping Locals 242
Using ROM Libraries 242
Suppressing Initialization 242

15 How To ... 249

How To Initialize the Vector Table 249
Initializing the Vector Table in the SmartLinker Prm File 249
Initializing the Vector Table in the Assembly Source File Using a Relocatable Section 251
Initializing the Vector Table in the Assembly Source File Using an Absolute Section 254

16 Messages 257

Message Kinds 257

1

Introduction

This section describes the SmartLinker. The linker merges the various object files of an application into one file, a so-called absolute file (or .ABS file for short; the file is called *absolute file* because it contains absolute, not relocatable code) that can be converted to a Motorola S-Record or an Intel Hex file using the Burner program or loaded into the target using the Downloader/Debugger.

The Linker is a smart linker, i.e. it will only link those objects that are actually used by your application.

This linker is able to generate either HIWARE or ELF absolute files.

For compatibility purpose, the HIWARE input syntax is also supported when ELF absolute files are generated.

Notation

Throughout this document, features or syntax which are only supported when ELF/ Dwarf absolute files are generated will be followed by (ELF) .

Features or syntax which are only supported when HIWARE absolute files are generated will be followed by (HIWARE) .

Features or syntax which are supported when either HIWARE or ELF absolute files are generated will be followed by (HIWARE+ELF) .

Structure of this Document

- **User interface**
- **Environment**
- **SmartLinker Options**: detailed description of the full set of Linker options
- **SmartLinker Commands**: list of all directives supported by the linker
- **SmartLinker Messages**: description with examples of messages produced by the SmartLinker

- Appendix
- Index

Purpose of a Linker

Linking is the process of assigning memory to all global objects (functions, global data, strings and initialization data) needed for a given application and combining these objects into a format suitable for downloading into a target system or an emulator.

The Linker is a smart linker: it only links those objects that are actually used by the application. Unused functions and variables won't occupy any memory in the target system. Besides this, there are other optimizations leading to low memory requirements of the linked program: initialization parts of global variables are stored in compact form and for equal strings, memory is reserved only once.

The most important features supported by the SmartLinker are:

- Complete control over the placement of objects in memory: it is possible to allocate different groups of functions or variables to different memory areas (Segmentation, please see section *Segments*).
- Linking to objects already allocated in a previous link session (ROM libraries).

NOTE User defined startup: The code for application startup is a separate file written in inline assembly and can be easily adapted to your particular needs. In this chapter and associated examples, the startup file is called startup.c / startup.o. However, this is a generic file name that has to be replaced by the real target startup file name given in the \LIB\COMPILER directory, in the README.TXT file (usually start*.c / start*.o where * is the name or a part of the MCU name and might also contain an abbreviation of the memory model). Please see also the README.TXT file or the STARTUP.TXT file in the \LIB\COMPILER directory for more details about memory models and associated startup codes.

- Mixed language linking: Modula-2, Assembly and C object files can be mixed, even in the same application.
- Initialization of vectors.

2

User Interface

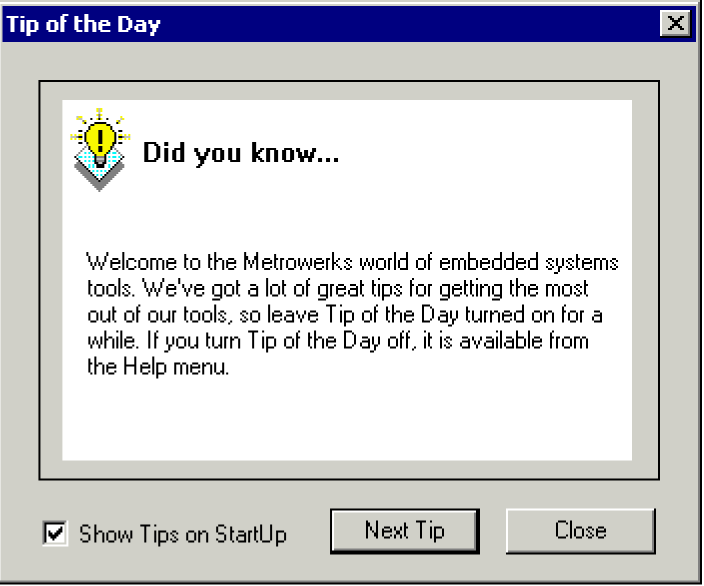
The SmartLinker runs under Win32.

Run the linker from the Shell, clicking the *Linker* icon on the shell tool bar.

Tip of The Day Dialog

When you start the SmartLinker, a standard *Tip of the Day* window is opened containing the last news about the SmartLinker, as shown in Figure 2.1.

Figure 2.1 Tip of the Day Window



The *Next Tip* button allows you to see the next tip about the SmartLinker.

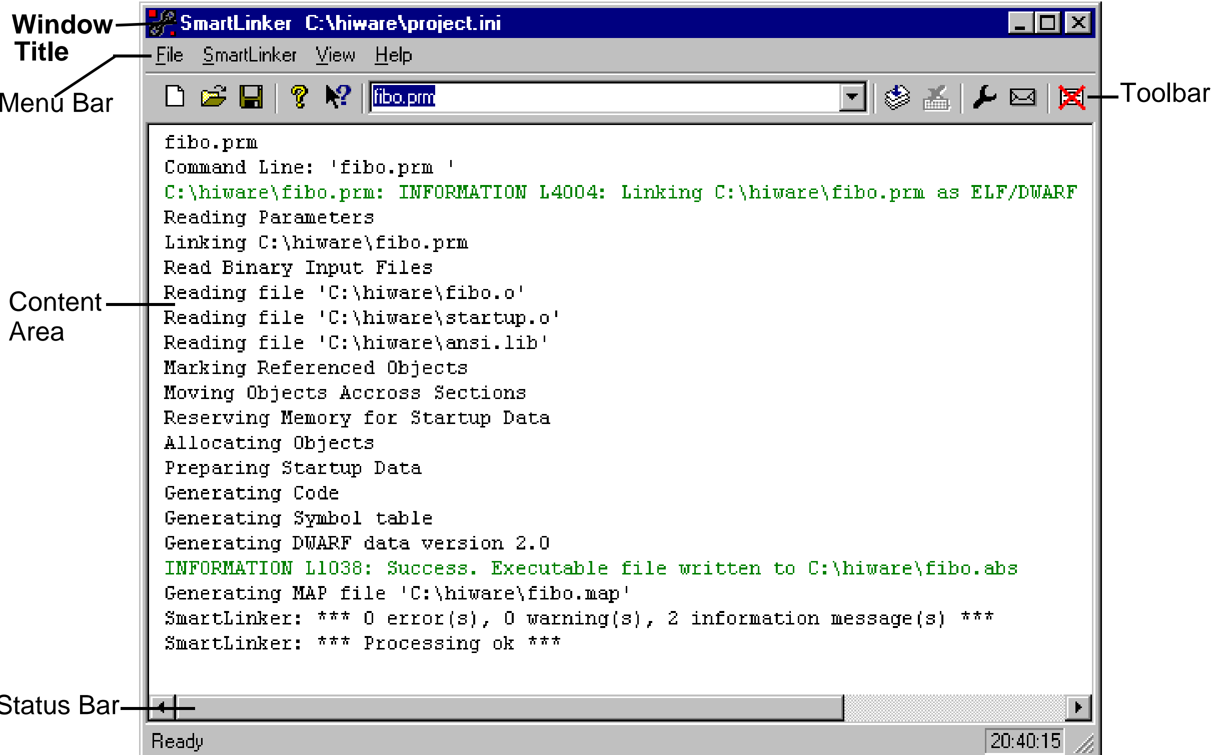
If you do not want to open automatically the standard *Tip of the Day* window when the SmartLinker is started, just unchecked the check box *Show Tips on StartUp*. Note that this configuration entry is stored in the local project file.

If you want to enable automatic display from the standard *Tip of the Day* window when the SmartLinker is started, just select the entry *Help | Tip of the Day*.... The window will be opened and you can check the box *Show Tips on StartUp*.

Click *Close* to close the *Tip of the Day* window.

Main Window

Figure 2.2 SmartLinker Main Window



This window is only visible on the screen when you do not specify any file name while starting the SmartLinker.

The SmartLinker window provides a window title, a menu bar, a tool bar, a content area and a status bar, as shown in Figure 2.2.

Window Title

The window title displays the project name. If currently no project is loaded, “Default Configuration” is displayed. A “*” after the configuration name indicates if some

values have been changed. The “*” appears as soon as an option, the editor configuration or the window appearance changes.

Content Area

The Content Area is used as a text container where logging information about the link session is displayed. This logging information consists of:

- the name of the prm file which is being linked.
- the whole name (including full path specification) of the files building the application.
- the list of the errors, warnings and information messages generated.

When a file name is dropped into the SmartLinker Window content area, the corresponding file is either loaded as configuration or linked. It is loaded as configuration if the file has the extension “ini”. If not, the file is linked with the current option settings (See *Specifying the Input File* below).

All text in the SmartLinker window content area can have context information. The context information consists of two items:

- a file name including a position inside of a file
- a message number

File context information is available for all output lines where a file name is displayed. There are two ways to open the file specified in the file context information in the editor specified in the editor configuration:

- If a file context is available for a line, double clicking on a line containing file context information.
- Click with the right mouse at a line and select “Open ..”. This entry is only available if a file context is available.

If a file can not be opened although a context menu entry is present, the editor configuration information is not correct (see the section Edit Settings Dialog below).

The message number is available for any message output. Then there are three ways to open the corresponding entry in the help file.

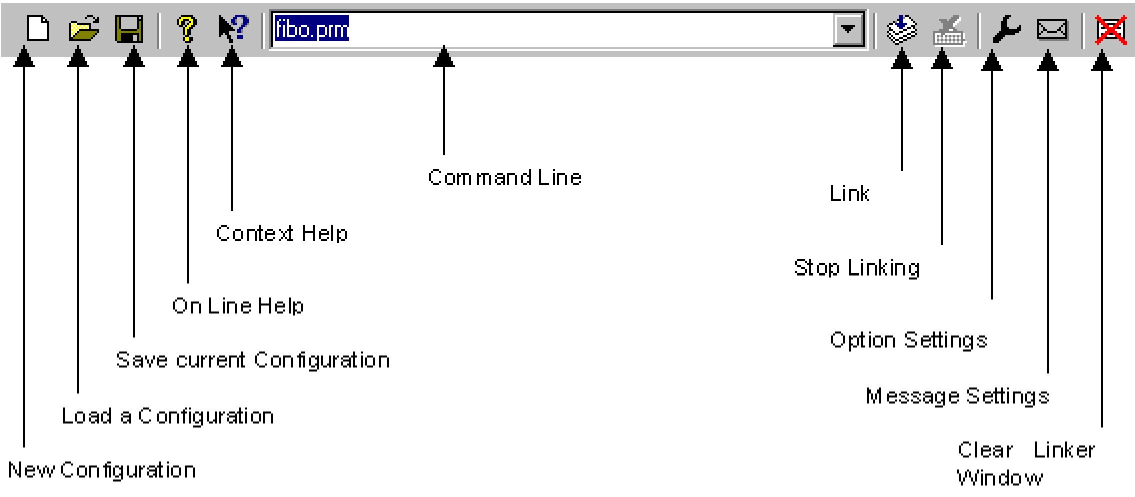
- Select one line of the message and press F1. If the selected line does not have a message number, the main help is displayed.
- Press Shift-F1 and then click on the message text. If the point clicked at does not have a message number, the main help is displayed.
- Click with the right mouse at the message text and select “Help on ...”. This entry is only available if a message number is available.

Messages are colored according to their kind. Errors are shown red, Fatal Errors dark red, Warnings blue and Information Messages green.

Tool Bar

Figure 2.3 describes the tool Bar buttons.

Figure 2.3 Tool Bar buttons



The three buttons on the left are linked with the corresponding entries of the *File* menu. The *New Configuration* , the *Load Configuration* and the *Save Configuration* allow to reset, load and save configuration files for the linker.

The *Help* button and the *Context Help* button allow to open the Help file or the Context Help.

When pressing , the mouse cursor changes it's form and has now a question mark beside the arrow. The help is called for the next item that is clicked. Specific help on Menus, toolbar buttons or on the window area can be get using the Context Help.

The command line history contains the list of the last commands executed. Once a command line has been selected or entered in this combo box, click the *Link* button to execute this command. The *Stop Linking* button allows to abort the current link session. If no link session is running, this button is disabled (gray).

The *Option Settings* button allows to open the *Option Settings* dialog.

The *Message Settings* button allows to open the *Message Settings* dialog.

The *Clear* button allows to clear the SmartLinker window content area.

The command line in the toolbar can be activated using the F2 key.

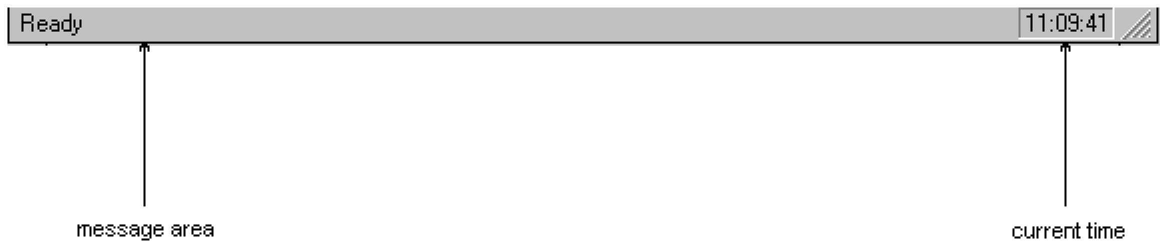
With the right mouse button, a context menu can be shown.

Messages are colored according to their Message Class.

Status Bar

Figure 2.4 shows the Status bar.

Figure 2.4 Status bar



When pointing to a button in the tool bar or a menu entry, the message area will display the function of the button or menu entry you are pointing to.

Linker Menu Bar

Following menus are available in the menu bar:

File	: Contains entries to manage SmartLinker configuration files.
SmartLinker	: Contains entries to set SmartLinker options.
View	: Contains entries to customize the SmartLinker window output.
Help	: A standard Windows Help menu.

File Menu

With the File Menu SmartLinker configuration files can be saved or loaded. A SmartLinker configuration file contains following information:

- the SmartLinker option settings specified in the SmartLinker dialog boxes
- the Message settings which specify which messages to display and which to treat as error.
- the list of the last command line executed and the current command line.
- the window position, size and font.
- the Tips of the Day settings, including if enabled at startup and which is the current entry
- Configuration files are text files, which have standard extension .ini. The user can define as many configuration files as required for his project, and can switch between the different configuration files using the *File | Load Configuration* and *File | Save Configuration* menu entry or the corresponding tool bar buttons. Table 2.1 describes the menu items with their description.

Table 2.1 File menu items and their description

Menu Item	Description
Link	Opens a standard Open File box, displaying the list of all the .prm files in the project directory. The input file can be selected using the features from the standard *Open File* box. The selected file will be linked as soon as the open File box is closed using *OK*.
New/Default Configuration	Resets the SmartLinker option settings to the default value. The SmartLinker options, which are activated per default, are specified in section *Command Line Options* from this document.
Load Configuration	Opens a standard Open File box, displaying the list of all the .INI files in the project directory. The configuration file can be selected using the features from the standard *Open File* box. The configuration data stored in the selected file is loaded and will be used by a further link session.
Save Configuration	Saves the current settings in the configuration file specified on the title bar.
Save Configuration as...	Opens a standard Save As box, displaying the list of all the .INI files in the project directory. The name or location of the configuration file can be specified using the features from the standard *Save As* box. The current settings are saved in the specified file as soon as the *save As* box is closed clicking *OK*

Table 2.1 File menu items and their description (*continued*)

Menu Item	Description
Configuration...	Opens the *Configuration* dialog box to specify the editor used for error feedback and which parts to save with a configuration.
1. .... project.ini *2. ....*	Recent project list. This list can be accessed to open a recently opened project again.
Exit	Closes the SmartLinker.

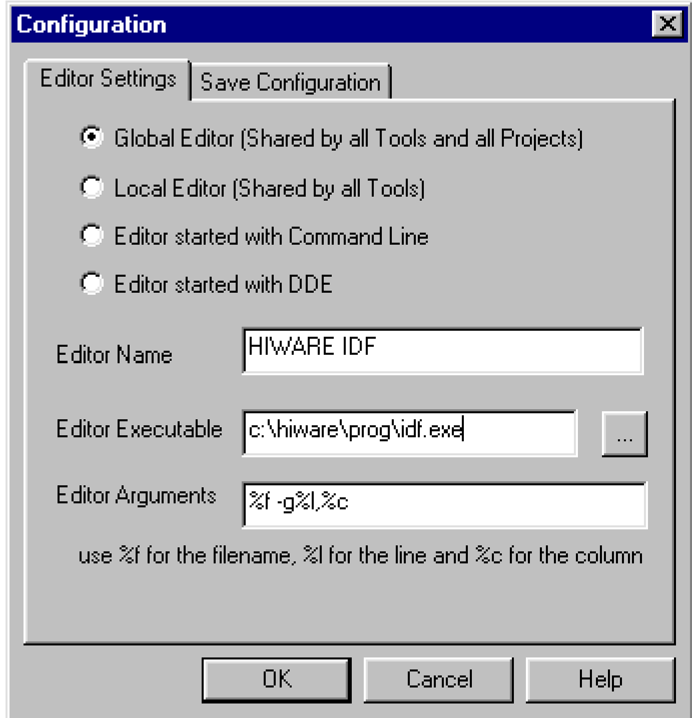
Edit Settings Dialog

The Editor Settings dialog box, as shown in Figure 2.5 has a main selection entry. Depending on the main type of editor selected, the content below changes.

There are the following main entries:

- Global Editor (Figure 2.5)

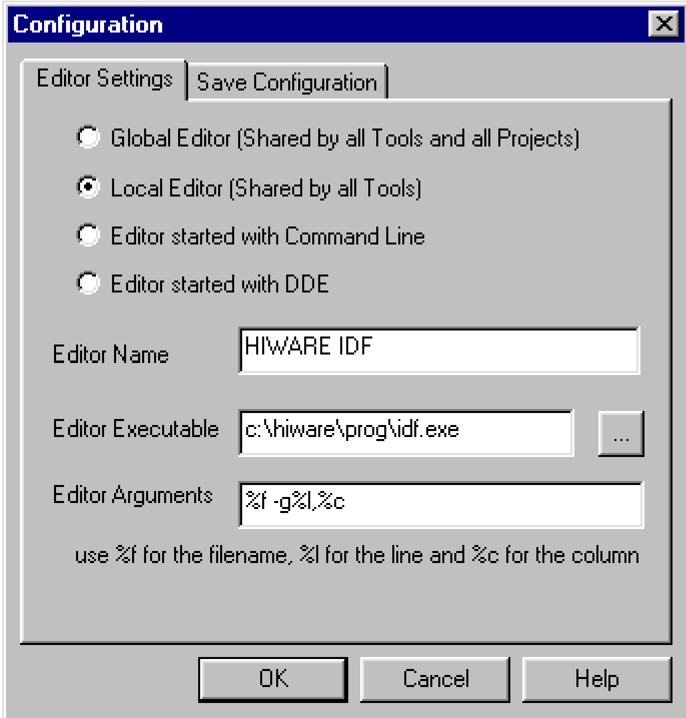
Figure 2.5 Editor Settings - Global Editor



The global editor is shared among all tools and projects on one computer. It is stored in the global initialization file "MCUTOOLS.INI" in the "[Editor]" section. Some Modifiers can be specified in the editor command line.

- Local Editor (Figure 2.6)

Figure 2.6 Editor Settings - Local Editor

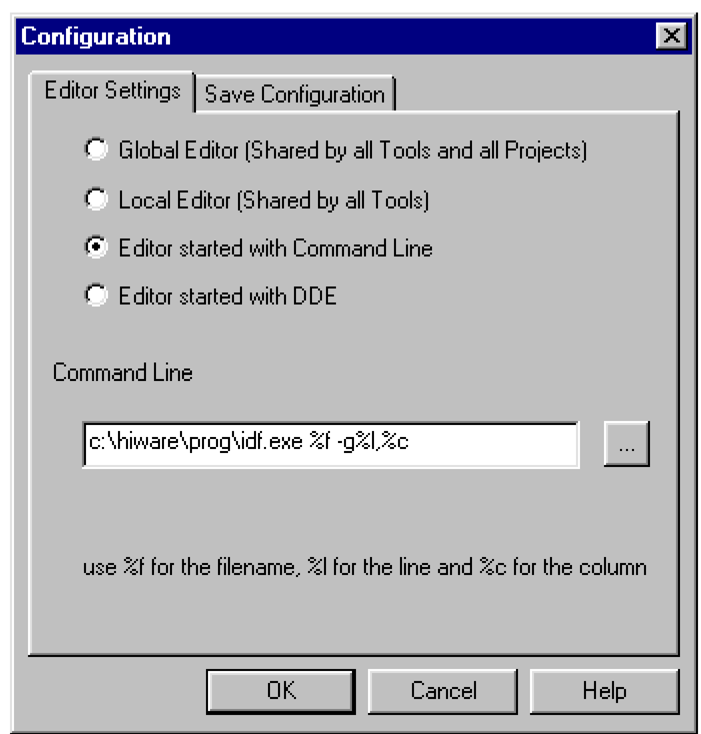


The local editor is shared among all tools using the same project file. Some Modifiers can be specified in the editor command line

The Global and Local Editor configuration can be edited with the linker. However, when these entries are stored, the behavior of the other tools using the same entry does also change when they are started the next time.

- Editor started with Command Line (Figure 2.7)

Figure 2.7 Editor Settings - Editor started with Command Line



When this editor type is selected, a separate editor is associated with the SmartLinker for error feedback. The editor configured in the Shell is not used for error feedback.

Enter the command, which should be used to start the editor.

The format from the editor command depends on the syntax, which should be used to start the editor. Some Modifiers can be specified in the editor command line to refer to a file name of a line number (See section Modifiers below).

The format from the editor command depends on the syntax which should be used to start the editor.

Example: (also look at the notes below)

For Winedit 32 bit version use (with an adapted path to the winedit.exe file)

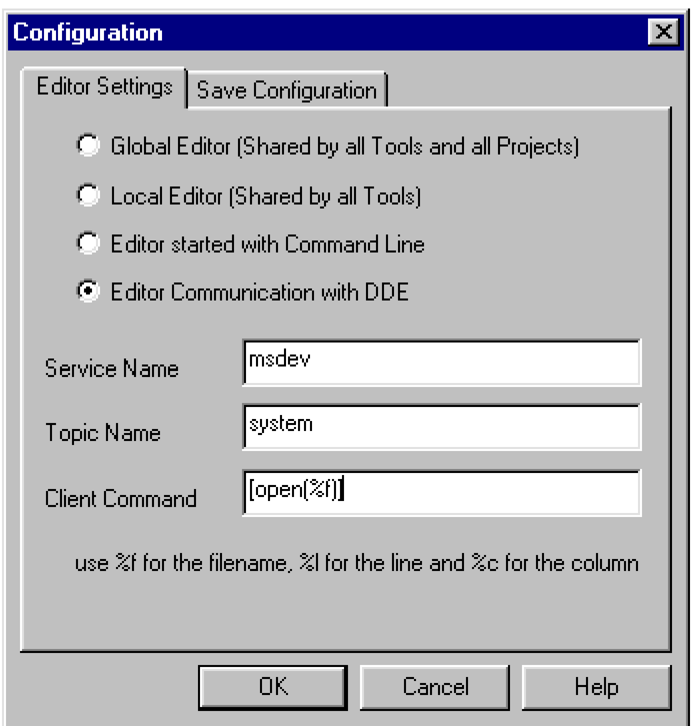
```
C:\WinEdit32\WinEdit.exe %f /#:%l
```

For Write.exe use (with an adapted path to the write.exe file, note that write does not support line number).

```
C:\Winnt\System32\Write.exe %f
```

- Editor Communication with DDE (Figure 2.8)

Figure 2.8 Editor Settings - Editor Communication with DDE



Enter the service, topic and client name to be used for a DDE connection to the editor. All entries can have modifiers for file name and line number as explained below in section Modifiers.

Example:

For Microsoft Developer Studio use the following setting:

```
Service Name: "msdev"
    Topic Name: "system"
    ClientCommand: "[open(%f)]"
```

- Modifiers

The configurations should contain some modifiers to tell the editor which file to open and at which line.

- The %f modifier refers to the name of the file (including path) where the error has been detected.
- The %l modifier refers to the line number where the message has been detected

---

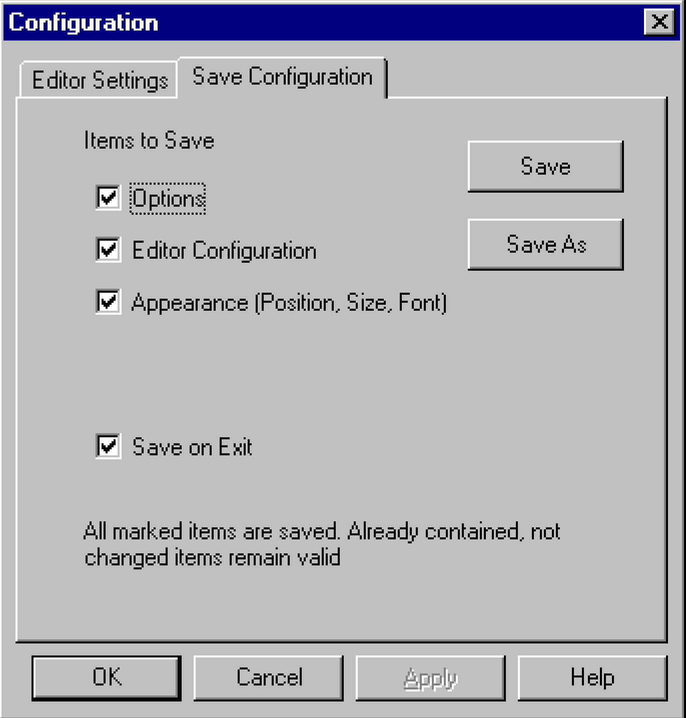
NOTE Be careful, the %l modifier can only be used with an editor which can be started with a line number as parameter. This is not the case for

WinEdit version 3.1 or lower or for the Notepad. When you work with such an editor, you can start it with the file name as parameter and then select the menu entry 'Go to' to jump on the line where the message has been detected. In that case the editor command looks like: `C:\WINAPPS\WINEDIT\Winedit.EXE %f`
Please check your editor manual to define the command line which should be used to start the editor.

Save Configuration Dialog

On the second index of the configuration dialog (Figure 2.9), all options considering the save operation are contained.

Figure 2.9 Save Configuration



In the *Save Configuration* index, four checkboxes allow to choose which items to save into a project file while the configuration is saved.

This dialog has the following configurations:

- Options: This item is related to the option and message settings. If this checkbox is set, the current option and message settings are stored in the project file when

the configuration is saved. By disabling this checkbox, changes done to the option and message settings are not saved, the previous settings remain valid.

- Editor Configuration: This item is related to the editor settings. If this checkbox is set, the current editor settings are stored in the project file when the configuration is saved. By disabling this checkbox, the previous settings remain valid.
- Appearance: This item is related to many parts like the window position (only loaded at startup time) and the command line content and history. If this checkbox is set, these settings are stored in the project file when the current configuration is saved. By disabling this checkbox, the previous settings remain valid.

NOTE By disabling selective options, only some parts of a configuration file can be written. For example when the suitable editor has been configured, the save Editor mark can be removed. Then future save commands will not modify the options any more.

- Save on Exit: If this option is set, the linker writes the configuration on exit. No question will appear to confirm this operation. If this option is not set, the linker does not write the configuration at exit, even if options or another part of the configuration has changed. No confirmation will appear in any case when closing the linker.

NOTE Almost all settings are stored in the project configuration file only. The only exceptions are:
- The recently used configuration list.
- All settings in this dialog.

NOTE The configurations of the linker can, and in fact are intended to, coexist in the same file as the project configuration of the shell. When the shell configures an editor, the linker can read this content out of the project file, if present. The project configuration file of the shell is named project.ini. This file name is therefore also suggested (but not mandatory) to the linker

SmartLinker Menu

The SmartLinker menu allows you to customize the SmartLinker. You can graphically set or reset SmartLinker options or define the optimization level you want to reach. Table 2.2 describes the SmartLinker menu items with their description.

Table 2.2 SmartLinker menu items and their description

Menu Item	Description
Options...	allows you to define the options which must be activated when linking an input file (See *Option Settings Dialog Box* below)
Messages	opens a dialog box, where the different error, warning or information messages can be mapped to another message class (See *Message Setting Dialog Box* below).
Stop Linking	stops the currently running linking process. This entry is only enabled (black) when a link process currently takes place. Otherwise, it is gray.

View Menu

The View menu allows you to customize the linker window. You can define if the status bar or the tool bar must be displayed or hidden. You can also define the font used in the window or clear the window. Table 2.3 describes the View menu items with their description.

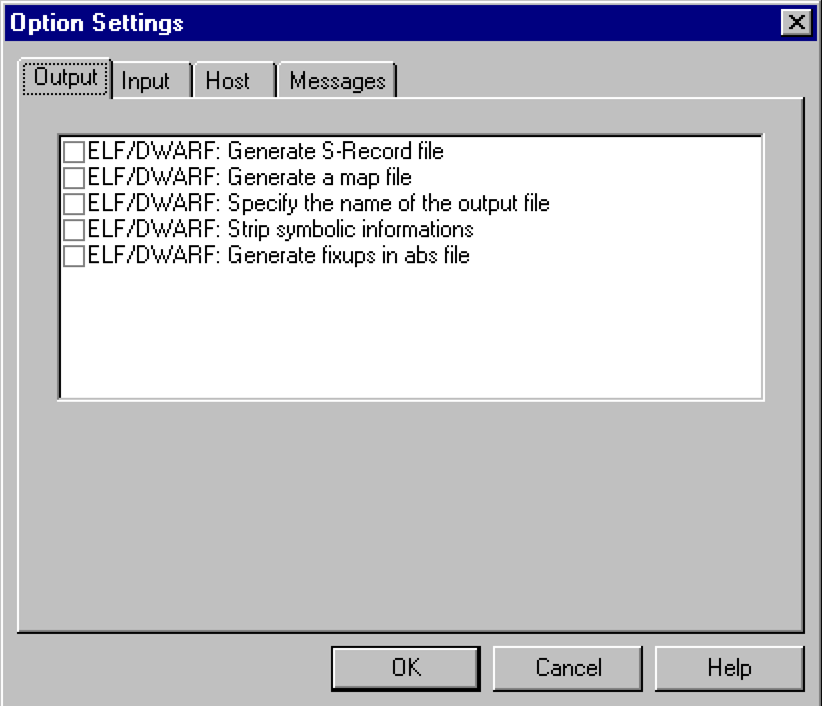
Table 2.3 View menu items and their description

Menu Item	Description
Tool Bar	switches display from the tool bar in the SmartLinker window.
Status Bar	switches display from the status bar in the SmartLinker window.
Log...	allows you to customize the output in the SmartLinker window content area. Following entries are available when *Log...* is selected:
Change Font	opens a standard font selection box. The options selected in the font dialog box are applied to the SmartLinker window content area.
Clear Log	allows you to clear the SmartLinker window content area.

Options Settings Dialog Box

The Options Settings dialog box (Figure 2.10) allows you to set/reset SmartLinker options.

Figure 2.10 Option Settings dialog box



The options available are arranged into different groups, and a sheet is available for each of these groups. The content of the list box depends on the selected sheet. Table 2.4 describes the groups and their description.

Table 2.4 Option Settings group and their description

Group	Description
Output	lists options related to the output files generation (what kind of files are to be generated).
Input	lists options related to the input files.
Messages	lists options controlling the generation of error messages.
Host	lists host specific options.

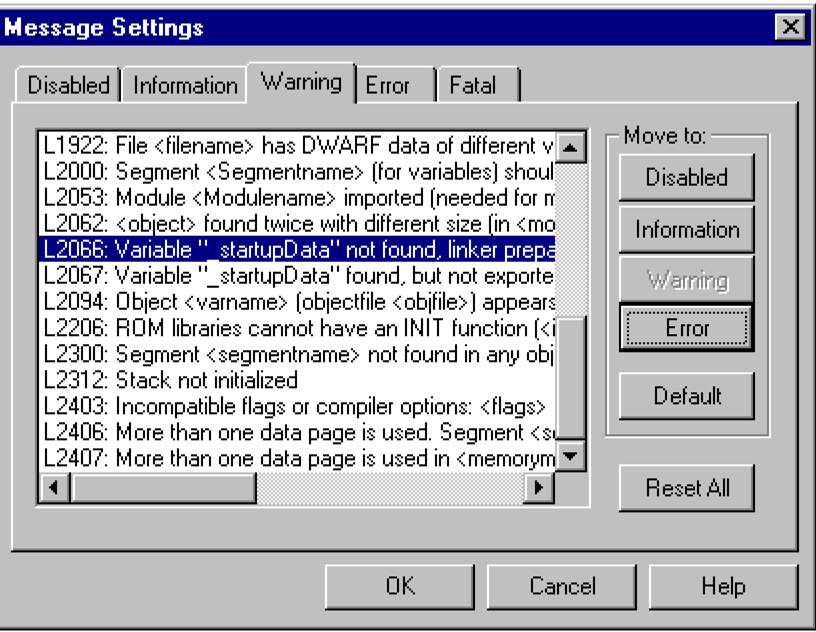
A SmartLinker option is set when its check box in front of it is checked. To obtain more detailed explanation about a specific option, select the option and the press the key F1 or the help button. To select an option, click once on the option text. The option text is then displayed inverted.

When the dialog is opened, then no option is yet selected. Pressing the key F1 or the help button then shows the help to this dialog.

Message Settings Dialog Box

The Message Settings (Figure 2.11) dialog box allows you to map messages to a different message class.

Figure 2.11 Message Settings dialog box



A sheet is available for each error message class and the content of the list box depends on the selected sheet. Table 2.5 describes the message classes available in the Message Settings dialog box.

Table 2.5 Message Class description

Message Class	Description	Color
Disabled	lists all messages disabled. That means messages displayed in the list box will not be displayed by the SmartLinker.	none.
Information	lists all information messages. Information messages informs about action taken by the SmartLinker.	green
Warning	lists all warning messages. When such a message is generated, linking of the input file continues and an absolute file is generated.	blue
Error	lists all error messages. When such a message is generated, linking of the input file continues but no absolute file is generated.	red
Fatal	lists all fatal error messages. When such a message is generated, linking of the input file stops immediately. Fatal messages can not be changed. There are only listed to call context help.	dark red

Depending on the message class, messages are shown in a different color in the main output area.

Each message has its own character (‘L’ for SmartLinker message) followed by a 4-5 digit number. This number allows an easy search for the message both in the manual or on-line help.

Changing the Class associated with a Message

You can configure your own mapping of messages in the different classes. In that purpose, you can use one of the buttons located on the right hand of the dialog box. Each button refers to a message class. To change the class associated with a message, you have to select the message in the list box and then click the button associated with the class where you want to move the message.

Example:

To define the message '*L1201: No stack defined*' (warning message) as an error message:

- Click the *Warning* sheet, to display the list of all warning messages in the list box.
- Click on the string '*L1201: No stack defined'* in the list box to select the message.
- Click *Error* to define this message as an error message.

NOTE	Messages cannot be moved from or to the fatal error class.

NOTE	The 'move to' buttons are only active when all selected messages can be moved. When one message is marked which cannot be moved to a specific group, the corresponding 'move to' button is disabled (grayed).

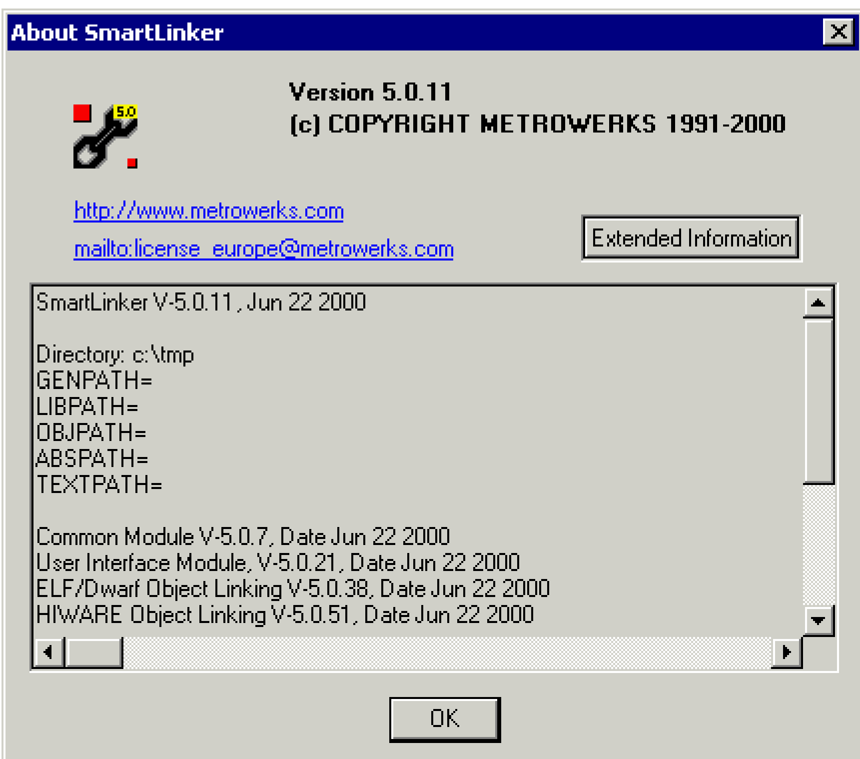
If you want to validate the modification you have performed in the error message mapping, close the 'Message settings' dialog box with the 'OK' button. If you close it using the 'Cancel' button, the previous message mapping remains valid.

To reset some messages to their default, select them and click on the 'Default' button. To reset all messages to the default, click on the 'Reset All' button.

About Box

The About box (Figure 2.12) can be opened with the Help->About command.

Figure 2.12 The About box



The about box contains many information. Among others, the current directory and the versions of subparts of the linker are shown. The main linker version is displayed separately on top of the dialog.

In addition, the about box contains all information needed to create a permanent license. The content of the about box can be used by copy and paste. Select the information, press the right mouse button and select “Copy”.

Click on OK to close this dialog.

During a linking session, the subversions of the linker parts can not be requested. They are only displayed if the linker currently is not processing.

Retrieving Information about an Error Message

You can access information about each message displayed in the list box. Select the message in the list box and then click *Help* or the F1 key. An information box is opened, which contains a more detailed description of the error message as well as a small example of code producing it. If several messages are selected, the help of the

first is shown. When no message is selected, pressing the key F1 or the help button shows the help for this dialog.

Specifying the Input File

There are different ways to specify the input file, which must be linked. During linking of a source file, the options are set according to the configuration performed by the user in the different dialog boxes and according to the options specified on the command line

Before starting to link a file make sure, you have associated a working directory with your linker.

Use the Command Line in the Tool Bar to Link

Linking a New File

A new file name and additional SmartLinker options can be entered in the command line. The specified file will be linked as soon as the button *Link* in the tool bar is selected or the enter key is pressed.

Linking a File which has already been linked

The command executed previously can be displayed using the arrow on the right side of the command line. A command is selected by clicking on it. It appears in the command line. The specified file will be linked as soon as the button *Link* in the tool bar is selected.

Use the Entry File | Link...

When the menu entry *File | Link...* is selected a standard file open file box is opened, displaying the list of all the prm file in the project directory. The user can browse to get the name of the file he wants to link. Select the desired file. Click Open in the Open File box to link the selected file.

Use Drag and Drop

A file name can be dragged from an external software (for example the File Manager/ Explorer) and dropped into the SmartLinker window. The dropped file will be linked as soon as the mouse button is released in the SmartLinker window. If a file being dragged has the extension “ini”, it is considered a configuration and it is immediately

loaded and not linked. To link a prm file with the extension “ini” use one of the other methods to link it.

Message/Error Feedback

After linking there are several ways to check where different errors or warnings have been detected. Per default, the format of the error message looks as follows:

```
>> <FileName>, line <line number>, col <column number>, pos <absolute
position in file>
<Portion of code generating the problem>
<message class><message number>: <Message string>
```

Example

```
>> in "placemen\tstpla8.prm", line 23, col 0, pos 668
  fpm_data_sec              INTO   MY_RAM2;

END

^
ERROR L1110: MY_RAM2 appears twice in PLACEMENT block
```

See also SmartLinker options -WMsgFi, -WMsgFb, -WMsgFob, -WMsgFoi, -WMsgFonF and -WMsgFonP for different message formats.

Use Information from the SmartLinker Window

Once a file has been linked, the SmartLinker window content area displays the list of all the errors or warnings detected.

The user can use his usual editor, to open the source file and correct the errors.

Use a User Defined Editor

The editor for *Error Feedback* must first be configured in the *Configuration* dialog box. The way error feedback is performed differently, depending if the editor can be started with a line number or not.

Line Number Can be Specified on the Command Line

Editor like WinEdit V95 or Higher or Codewright can be started with a line number in the command line. When these editors have been correctly configured, they can be activated automatically by double clicking on an error message. The configured editor will be started, the file where the error occurs is automatically opened and the cursor is placed on the line where the error was detected.

Line Number Cannot be Specified on The Command Line

Editor like WinEdit V31 or lower, Notepad, Wordpad cannot be started with a line number in the command line. When these editors have been correctly configured, they can be activated automatically by double clicking on an error message. The configured editor will be started, the file where the error occurs is automatically opened. To scroll to the position where the error was detected, you have to:

- Activate the assembler again
- Click the line on which the message was generated. This line is highlighted on the screen.
- Copy the line in the clipboard pressing CTRL + C
- Activate the editor again.
- Select *Search |Find*, the standard Find dialog box is opened.
- Copy the content of the clipboard in the Edit box pressing CTRL + V
- Click *Forward* to jump to the position where the error was detected.

3 Environment

This part of the document describes the environment variables used by the SmartLinker. Some of those environment variables are also used by other tools (for example, Macro Assembler, Compiler, ...), so consult also their respective manual.

Various parameters of the SmartLinker may be set in an environment using so-called environment variables. The syntax is always the same:

```
Parameter = KeyName "=" ParamDef.
```

NOTE *No* blanks are allowed in the definition of an environment variable

Example:

```
GENPATH=C:\INSTALL\LIB;D:\PROJECTS\TESTS;/usr/local/lib;/
home/me/my_project
```

These parameters may be defined in several ways:

- Using system environment variables supported by your operating system.
- Putting the definitions in a file called DEFAULT.ENV (.hidefaults for UNIX) in the project directory.
- Putting the definitions in a file given by the value of the system environment variable ENVIRONMENT

NOTE The project directory mentioned above can be set via the system environment variable `DEFAULTDIR`

When looking for an environment variable, all programs first search the system environment, then the DEFAULT.ENV (.hidefaults for UNIX) file and finally the global environment file given by ENVIRONMENT. If no definition can be found, a default value is assumed.

NOTE The environment may also be changed using the -Env SmartLinker option.

The Current Directory

The most important environment for all tools is the current directory. The current directory is the base search directory where the tool starts to search for files (for example, for the DEFAULT.ENV / .hidefaults)

Normally, the current directory of a tool started is determined by the operation system or by the program who launches another one (for example, WinEdit).

For the UNIX operating system, the directory in which, an executable is started is also the current directory from where the binary file has been started.

For MS Windows based operating systems, the current directory definition is quite complex:

- If the tool is launched using a File Manager/Explorer, the current directory is the location of the executable launched.
- If the tool is launched using an Icon on the Desktop, the current directory is the working directory specified and associated with the Icon.
- If the tool is launched by dragging a file on the icon of the executable under Windows 95 or Windows NT 4.0, the desktop is the current directory.
- If the tool is launched by another launching tool with its own working directory specification (e.g. an editor as WinEdit), the current directory is the one specified by the launching tool (e.g. working directory definition in WinEdit).
- Changing the current project file does also change the current directory if the other project file is in a different directory. Note that browsing for a prm file does not change the current directory.

To overwrite this behavior, the environment variable DEFAULTDIR may be used.

The current directory is displayed among other information with the linker option “-v” and in the about box.

Global Initialization File (MCUTOOLS.INI) (PC only)

All tools may store some global data into the MCUTOOLS.INI. The tool first searches for this file in the directory of the tool itself (path of the executable). If there is no MCUTOOLS.INI file in this directory, the tool looks for a MCUTOOLS.INI file located in the MS Windows installation directory (for example, C:\WINDOWS).

Example:

`C:\WINDOWS\MCUTOOLS.INI`

`D:\INSTALL\PROG\MCUTOOLS.INI`

If a tool is started in the D:\INSTALL\PROG\DIRECTOY, the current file in the same directory than the tool is used `(D:\INSTALL\PROG\MCUTOOLS.INI)`.

However, if the tool is started outside the D:\INSTALL\PROG directory, the current file in the Windows directory is used `(C:\WINDOWS\MCUTOOLS.INI)`.

[Installation] Section

Entry: Path

Arguments: Last installation path.

Description: Whenever a tool is installed, the installation script stores the installation destination directory into this variable.

Example: Path=c:\install

Entry: Group

Arguments: Last installation program group.

Description: Whenever a tool is installed, the installation script stores the installation program group created into this variable.

Example: Group=ANSI-C Compiler

[Options] Section

Entry: DefaultDir

Arguments: Default Directory to be used.

Description: Specifies the current directory for all tools on a global level (see also environment variable DEFAULTDIR).

Example: DefaultDir=c:\install\project

[LINKER] Section

Entry: SaveOnExit

Arguments: 1/0

Description: 1 if the configuration should be stored when the linker is closed, 0 if it should not be stored. The linker does not ask to store a configuration in either cases.

Entry: SaveAppearance

Arguments: 1/0

Description: 1 if the visible topics should be stored when writing a project file, 0 if not. The command line, its history, the windows position and other topics belong to this entry.

Entry: SaveEditor

Arguments: 1/0

Description: 1 if the visible topics should be stored when writing a project file, 0 if not. The editor settings contain all information of the editor configuration dialog.

Entry: SaveOptions

Arguments: 1/0

Description: 1 if the options should be contained when writing a project file, 0 if not. The options do also contain the message settings.

Entry: RecentProject0, RecentProject1, ...

Arguments: names of the last and prior project files

Description: This list is updated when a project is loaded or saved. Its current content is shown in the file menu.

Entry: TipFilePos

Arguments: any integer, e.g. 236

Description: Index which tip is actually shown, used to display different tips every time.

Entry: ShowTipOfDay

Arguments: 0/1

Description: Should the Tip of the Day dialog be shown at startup.

1: it should be shown

0: no, only when opened in the help menu

Entry: TipTimeStamp

Arguments: date

Description: This entry is used to remark when a new tips are available. Whenever the date specified here does not match the date of the tips, the first tip is displayed.

Example:

```
[LINKER]
TipFilePos=357
TipTimeStamp=Jan 25 2000 12:37:41
ShowTipOfDay=0
SaveOnExit=1
SaveAppearance=1
SaveEditor=1
SaveOptions=0
RecentProject0=C:\myprj\project.ini
RecentProject1=C:\otherprj\project.ini
```

[Editor] Section

Entry: Editor_Name

Arguments: The name of the global editor

Description: Specifies the name, which is displayed for the global editor. This entry has only a description effect. Its content is not used to start the editor.

Saved: Only with Editor Configuration set in the File->Configuration Save Configuration dialog.

Entry:	Editor_Exe
Arguments:	The name of the executable file of the global editor
Description:	Specifies file name (including path), which is called for showing a text file, when the global editor setting is active. In the editor configuration dialog, the global editor selection is only active when this entry is present and not empty.
Saved:	Only with Editor Configuration set in the File->Configuration Save Configuration dialog.

Entry:	Editor_Opts
Arguments:	The options to use the global editor
Description:	Specifies options, which should be used for the global editor. If this entry is not present or empty, “%f” is used. The command line to launch the editor is build by taking the Editor_Exe content, then appending a space followed by this entry.
Saved:	Only with Editor Configuration set in the File->Configuration Save Configuration dialog.

Example: [Editor]

editor_name=WinEdit

editor_exe=C:\Winedit\WinEdit.exe

editor_opts=%f

Example

The following example shows a typical layout of the MCUTOOLS.INI:

```
[Installation]
Path=c:\metrowerks
Group=ANSI-C Compiler

[Editor]
editor_name=WinEdit
editor_exe=C:\Winedit\WinEdit.exe
editor_opts=%f

[Options]
DefaultDir=c:\myprj
```

```
[Linker]
SaveOnExit=1
SaveAppearance=1
SaveEditor=1
SaveOptions=1
RecentProject0=c:\myprj\project.ini
RecentProject1=c:\otherprj\project.ini
```

Local Configuration File (usually project.ini)

The SmartLinker does not change the default.env in any way. Its content is only read. All the configuration properties are instead stored in the configuration file. The same configuration file can and is intended to be used by different applications.

The shell uses the configuration file with the name “project.ini” in the current directory only, that is why this name is also suggested to be used with the linker. Only when the shell uses the same file as the linker, the linker can use the editor configuration written and maintained by the shell. Apart from this, the linker can use any file name for the project file. The configuration file does have the same format as windows ini files. The linker stores its own entries with the same section name as in the global mcutools.ini file. Different versions of the linker are using the same entries. This mainly plays a role when options only available in one version should be stored in the configuration file. In such situations, two files must be maintained for the different linker versions. If no incompatible options are enabled when the file is last saved, the same file may can be used for both linker version.

The current directory is always the directory, where the configuration is in. If a configuration file in a different directory is loaded, then the current directory also changes. When the current directory changes, also the whole default.env file is reloaded. Always when a configuration file is loaded or stored, the options in the environment variable LINKOPTIONS are reloaded and added to the project options. This behavior has to be noticed when in different directories different default.env exist which do contain incompatible options in the LINKOPTIONS. When a project is loaded using the first default.env, its LINKOPTIONS are added to the configuration file. If then this configuration is stored in a different directory, where a default.env exists with the incompatible options, the linker adds options and remarks the inconsistency. Then a message box appears to inform the user that the default.env options were not added. In such a situation, the user can either remove the option from the configuration file with the advanced option dialog or he can remove the option

from the default.env with the shell or a text editor depending which options should be used in the future.

At startup there are two ways to load a configuration:

- use the command line option -Prod
- the file project.ini the current directory

If the option -Prod is used, then the current directory is the directory the project file is in. If the option -prog is used with a directory, the file project.ini in this directory is loaded.

[Editor] Section

Entry:	Editor_Name
Arguments:	The name of the local editor
Description:	Specifies the name, which is displayed for the local editor. This entry has only a description effect. Its content is not used to start the editor.
Saved:	Only with Editor Configuration set in the File->Configuration Save Configuration dialog. This entry has the same format as for the global editor configuration in the mcutools.ini file.

Entry:	Editor_Exe
Arguments:	The name of the executable file of the local editor (including path).
Description:	Specifies file name with is called for showing a text file, when the local editor setting is active. In the editor configuration dialog, the local editor selection is only active when this entry is present and not empty.
Saved:	Only with Editor Configuration set in the File->Configuration Save Configuration dialog. This entry has the same format as for the global editor configuration in the mcutools.ini file.

Entry:	Editor_Opts
Arguments:	The options to use the local editor

Description: Specifies options, which should be used for the local editor. If this entry is not present or empty, “%f” is used. The command line to launch the editor is build by taking the Editor_Exe content, then appending a space followed by this entry.

Saved: Only with Editor Configuration set in the File->Configuration Save Configuration dialog.
This entry has the same format as for the global editor configuration in the mcutools.ini file.

Example: [Editor]
editor_name=WinEdit
editor_exe=C:\Winedit\WinEdit.exe
editor_opts=%f

[[LINKER] Section

Entry: RecentCommandLineX, X= Integer

Arguments: String with a command line history entry, for example, fibo.prm

Description: This list of entries contains the content of command line history.

Saved: Only with Appearance set in the File > Configuration Save Configuration dialog.

Entry: CurrentCommandLine

Arguments: String with the command line, for example, fibo.prm -w1

Description: The currently visible command line content.

Saved: Only with Appearance set in the File->Configuration Save Configuration dialog.

Entry: StatusbarEnabled

Arguments: 1/0

Special: This entry is only considered at startup. Later load operations do not use it any more.

Description: Is currently the status bar enabled.

1: the status bar is visible

0: the status bar is hidden

Saved: Only with Appearance set in the File > Configuration Save Configuration dialog.

Entry: ToolbarEnabled

Arguments: 1/0

Special: This entry is only considered at startup. Later load operations do not use it any more.

Description: Is currently the tool bar enabled.

1: the tool bar is visible

0: the tool bar is hidden

Saved: Only with Appearance set in the File > Configuration Save Configuration dialog.

Entry: WindowPos

Arguments: 10 integers, e.g. “0,1,-1,-1,-1,-1,390,107,1103,643”

Special: This entry is only considered at startup. Later load operations do not use it any more.
Changes of this entry do not show the “*” in the title.

Description: This numbers contain the position and the state of the window (maximized,..) and other flags.

Saved: Only with Appearance set in the File > Configuration Save Configuration dialog.

Entry: WindowFont

Arguments: size: == 0 -> generic size, < 0 -> font character height, > 0 font cell height

weight: 400 = normal, 700 = bold (valid values are 0..1000)

italic: 0 == no, 1 == yes

font name: max 32 characters.

Description: Font attributes.

Saved: Only with Appearance set in the File > Configuration Save Configuration dialog.

Example: WindowFont=-16,500,0,Courier

Entry: Options

Arguments: -W2

Description: The currently active option string. Because also the messages are be contained here, this entry can be very long.

Saved: Only with Options set in the File > Configuration Save Configuration dialog.

Entry: EditorType

Arguments: 0/1/2/3

Description: 0: global editor configuration (in the file mcutools.ini)
1: local editor configuration (the one in this file)
2: command line editor configuration, entry EditorCommandLine
3: DDE editor configuration, entries beginning with EditorDDE
For details see also Editor Configuration.

Saved: Only with Editor Configuration set in the File > Configuration Save Configuration dialog.

Entry: EditorCommandLine

Arguments: command line, for WinEdit: “C:\Winapps\WinEdit.exe %f /#:%l”

Description: Command line content to open a file. For details see also Editor Configuration.

Saved: Only with Editor Configuration set in the File > Configuration Save Configuration dialog.

Entry: EditorDDEClientName

Arguments: client commend, for example, “[open(%f)]”

Description: Name of the client for DDE editor configuration.
For details see also Editor Configuration.

Saved: Only with Editor Configuration set in the File > Configuration Save Configuration dialog.

Entry: EditorDDETopicName

Arguments: topic name, for example, “system”

Description: Name of the topic for DDE editor configuration.
For details see also Editor Configuration.

Saved: Only with Editor Configuration set in the File > Configuration Save Configuration dialog.

Entry: EditorDDEServiceName

Arguments: service name, for example, “system”

Description: Name of the service for DDE editor configuration.
For details see also Editor Configuration.

Saved: Only with Editor Configuration set in the File > Configuration Save Configuration dialog.

Example

The following example shows a typical layout of the configuration file (usually project.ini):

```
[Editor]
Editor_Name=WinEdit
Editor_Exe=C:\WinEdit\WinEdit.exe %f /#:%l
Editor_Opts=%f

[Linker]
StatusbarEnabled=1
ToolbarEnabled=1
WindowPos=0,1,-1,-1,-1,-1,390,107,1103,643
WindowFont=-16,500,0,Courier
Options=-w1
EditorType=3
RecentCommandLine0=fibo.prm -w2
RecentCommandLine1=fibo.prm
CurrentCommandLine=calc.prm -w2
EditorDDEClientName=[open(%f)]
EditorDDETopicName=system
EditorDDEServiceName=msdev
EditorCommandLine=C:\WinEdit\WinEdit.exe %f /#:%l
```

Paths

Most environment variables contain path lists telling where to look for files. A path list is a list of directory names separated by semicolons following the syntax below:

```
PathList = DirSpec {";" DirSpec}.
```

```
DirSpec  = ["*"] DirectoryName.
```

Example:

```
GENPATH=C:\INSTALL\LIB;D:\PROJECT\TEST;\usr\loc\metrowerks\lib
;\home\me
```

If a directory name is preceded by an asterisk (`"*"`), the programs recursively search that whole directory tree for a file, not just the given directory itself. The directories are searched in the order they appear in the path list.

Example:

```
LIBPATH=*C:\INSTALL\LIB
```

NOTE Some DOS/UNIX environment variables (like GENPATH, LIBPATH, etc.) are used. For further details refer to “Environment.”

Line Continuation

It is possible to specify an environment variable in a environment file (default.env/ .hidefautls) over different lines using the line continuation character ‘\’:

Example:

```
COMPOPTIONS=\
-W2 \
-Wpd
```

This is the same as

```
COMPOPTIONS=-W2 -Wpd
```

However, this feature may be dangerous using it together with paths, for example,

```
GENPATH=.\
TEXTFILE=.\txt
```

will result in

```
GENPATH=.TEXTFILE=.\txt
```

To avoid such problems, we recommend to use a semicolon';' at the end of a path if there is a '\' at the end:

```
GENPATH=.\;
TEXTFILE=.\txt
```

Environment Variable Details

The remainder of this section is devoted to describing each of the environment variables available for the SmartLinker. Table 3.1 contains options in alphabetical order and each is divided into several sections.

ABSPATH

Table 3.1 Environment variables and their description

Topic	Description
Tools	Lists tools which are using this variable
Synonym	For some environment variables, a synonym also exists. Those synonyms may be used for older releases of the SmartLinker and will be removed in the future. A synonym has lower precedence than the environment variable.
Syntax	Specifies the syntax of the option in a EBNF format.
Arguments	Describes and lists optional and required arguments for the variable.
Default	Shows the default setting for the variable or none.
Description	Provides a detailed description of the option and how to use it.
Example	Gives an example of usage, and effects of the variable where possible. The example shows an entry in the default.env for PC or in the .hidefaults for UNIX.
See also	Names related sections.

ABSPATH: Absolute Path

Tools: SmartLinker, Debugger

Synonym: None

Syntax: "ABSPATH=" {<path>}.

Arguments: <path>: Paths separated by semicolons, without spaces.

Description: When this environment variable is defined, the SmartLinker will store the absolute files it produces in the first directory specified there. If ABSPATH is not set, the generated absolute files will be stored in the directory the parameter file was found.

Example: ABSPATH=\sources\bin;..\..\headers;\usr\local\bin

See also: none

COPYRIGHT

COPYRIGHT: Copyright Entry in Absolute File

Tools: Compiler, Assembler, SmartLinker, Librarian

Synonym: none.

Syntax: "COPYRIGHT=" <copyright>.

Arguments: <copyright>: copyright entry.

Default: none.

Description: Each absolute file contains an entry for a copyright string. This information may be retrieved from the absolute files using the decoder.

Example: COPYRIGHT=Copyright by PowerUser

See also: Environment variable USERNAME
Environment variable INCLUDETIME

DEFAULTDIR

DEFAULTDIR: Default Current Directory

Tools: Compiler, Assembler, SmartLinker, Decoder, Debugger, Librarian, Maker, Burner

Synonym: none.

Syntax: "DEFAULTDIR=" <directory>.

Arguments: <directory>: Directory to be the default current directory.

Default: none.

Description: With this environment variable the default directory for all tools may be specified. All the tools indicated above will take the directory specified as their current directory instead the one defined by the operating system or launching tool (e.g. editor).

NOTE This is an environment variable on system level (global environment variable) It cannot be specified in a default environment file (DEFAULT.ENV/.hidefaults)

Example: DEFAULTDIR=C:\INSTALL\PROJECT

See also: Section The Current Directory
Section MCUTOOLS.INI File'

ENVIRONMENT

ENVIRONMENT: Environment File Specification

Tools: Compiler, SmartLinker, Decoder, Debugger, Librarian, Maker

Synonym: HIENVIRONMENT

Syntax: "ENVIRONMENT=" <file>.

Arguments: <file>: file name with path specification, without spaces

Default: none.

Description: This variable has to be specified on system level. Normally the SmartLinker looks in the current directory for a environment file named default.env (.hidefaults on UNIX). Using ENVIRONMENT (e.g. set in the autoexec.bat (DOS) or .cshrc (UNIX)), a different file name may be specified.

NOTE This is an environment variable on system level (global environment variable) It cannot be specified in a default environment file (DEFAULT.ENV/.hidefaults)

Example: ENVIRONMENT=\METROWERKS\prog\global.env

See also: none.

ERRORFILE

ERRORFILE: Error File Name Specification

Tools:	Compiler, SmartLinker, Assembler
Synonym:	none.
Syntax:	"ERRORFILE=" <filename>.
Arguments:	<filename>: File name with possible format specifiers.
Description:	The environment variable ERRORFILE specifies the name for the error file (used by the SmartLinker). Possible format specifiers are: '%n': Substitute with the file name, without the path. '%p': Substitute with the path of the source file. '%f': Substitute with the full file name, i.e. with the path and name (the same as '%p%n'). In case of an illegal error file name, a notification box is shown.

Example: ERRORFILE=MyErrors.err
lists all errors into the file MyErrors.err in the project directory.

ERRORFILE=\tmp\errors

lists all errors into the file errors in the directory \tmp.

ERRORFILE=%f.err

lists all errors into a file with the same name as the source file, but with extension .err, into the same directory as the source file, for example, if we link a file \sources\test.prm, an error list file \sources\test.err will be generated.

ERRORFILE=\dir1\%n.err

For example, for a source file test.prm, an error list file \dir1\test.err will be generated.

ERRORFILE=%p\errors.txt

For example, for a source file \dir1\dir2\test.prm, an error list file \dir1\dir2\errors.txt will be generated.

If the environment variable ERRORFILE is not set, the errors are written to the file EDOUT in the project directory.

If the environment variable ERRORFILE is not set, errors are written to the default error file.

The default error file name depends on the way the linker is started.

If a file name is provided on the linker command line, the errors are written to the file EDOUT in the project directory.

If no file name is provided on the linker command line, the errors are written to the file ERR.TXT in the project directory.

See also: none.

GENPATH

GENPATH: Define Paths to search for input Files

Tools: Compiler, Assembler, SmartLinker, Decoder, Debugger

Synonym: HIPATH

Syntax: "GENPATH=" {<path>}.

Arguments: <path>: Paths separated by semicolons, without spaces.

Description: The SmartLinker will look for the prm first in the project directory, then in the directories listed in the environment variable GENPATH. The object and library files specified in the linker prm file are searched in the project directory, then in the directories listed in the environment variable OBJPATH and finally in those specified in GENPATH

NOTE If a directory specification in this environment variables starts with an asterisk ("*"), the whole directory tree is searched recursively depth first, i.e. all subdirectories and *their* subdirectories and so on are searched, too. Within one level in the tree, search order of the subdirectories is indeterminate.

Example: GENPATH=\obj;..\..\lib;

See also: none

INCLUDETIME

INCLUDETIME: Creation Time in Object File

Tools: Compiler, Assembler, SmartLinker, Librarian

Synonym: none.

Syntax: "INCLUDETIME=" ("ON" | "OFF").

Arguments: "ON": Include time information into object file.

"OFF": Do not include time information into object file.

Default: "ON"

Description: Normally each absolute file created contains a time stamp indicating the creation time and data as strings. So whenever a new file is created by one of the tools, the new file gets a new time stamp entry.

This behavior may be undesired if for SQA reasons a binary file compare has to be performed. Even if the information in two absolute files is the same, the files do not match exactly because the time stamps are different. To avoid such problems this variable may be set to OFF. In this case the time stamp strings in the absolute file for date and time are “none” in the object file.

The time stamp may be retrieved from the object files using the decoder.

Example: INCLUDETIME=OFF

See also: Environment variable COPYRIGHT

Environment variable USERNAME

LINKOPTIONS

LINKOPTIONS: Default SmartLinker Options

Tools: SmartLinker

Synonym: None

Syntax: "LINKOPTIONS=" {<option>}.

Arguments: <option>: SmartLinker command line option

Default: none.

Description: If this environment variable is set, the SmartLinker appends its contents to its command line each time a file is linked. It can be used to globally specify certain options that should always be set, so you do not have to specify them each time a file is linked.

Example: LINKOPTIONS=-W2

See also: SmartLinker options

OBJPATH

OBJPATH: Object File Path

Tools: Compiler, Assembler, SmartLinker, Decoder, Debugger

Synonym: None

Syntax: "OBJPATH=" {<path>}.

Arguments: <path>: Paths separated by semicolons, without spaces.

Description: When this environment variable is defined, the linker search for the object and library files specified in the linker prm file in the project directory, then in the directories listed in the environment variable OBJPATH and finally in those specified in GENPATH.

Example: OBJPATH=\sources\bin;..\..\headers;\usr\local\bin

See also: Option -L

RESETVECTOR

RESETVECTOR: Reset Vector Location

Tools: Compiler, Assembler, SmartLinker, Simulator for HC05 and St7 only

Synonym: None

Syntax: "RESETVECTOR=" <Address>.

Arguments: <Address>: Address of reset vector. The default is 0xFFFE.

Description: For the HC05 and the St7 architecture, the reset vector location depends on the actual derivative. For the VECTOR directive, the linker has to know where the VECTOR 0 has to be placed.

Example: RESETVECTOR=0xFFFE

See also: none

SRECORD

SRECORD: S Record File Format

Tools: Assembler, SmartLinker, Burner

Synonym: None

Syntax: "SRECORD=" <RecordType>.

Arguments: <Record Type>: Force the type for the Motorola S record which must be generated. This parameter may take the value ‘S1’, ‘S2’ or ‘S3’.

Description: This environment variable is only relevant when absolute files are directly generated by the macro assembler instead of object files. When this environment variable is defined, the Assembler will generate a Motorola S file containing records from the specified type (S1 records when S1 is specified, S2 records when S2 is specified and S3 records when S3 is specified).

NOTE If the environment variable SRECORD is set, it is the user responsibility to specify the appropriate S record type. If you specifies S1 while your code is loaded above 0xFFFF, the Motorola S file generated will not be correct, because the addresses will all be truncated to 2 bytes values.

When this variable is not set, the type of S record generated will depend on the size of the address loaded there. If the address can be coded on 2 bytes, a S1 record is generated. If the address is coded on 3 bytes, a S2 record is generated. Otherwise a S3 record is generated.

Example: SRECORD=S2

See also: none

TEXTPATH

TEXTPATH: Text Path

Tools: Compiler, Assembler, SmartLinker, Decoder

Synonym: None

Syntax: “TEXTPATH=” {<path>}.

Arguments: <path>: Paths separated by semicolons, without spaces.

Description: When this environment variable is defined, the SmartLinker will store the map file it produces in the first directory specified there. If TEXTPATH is not set, the generated map file will be stored in the directory the prm file was found.

Example: TEXTPATH=\sources..\..\headers;\usr\local\txt

See also: None

TMP

TMP: Temporary directory

Tools: Compiler, Assembler, SmartLinker, Debugger, Librarian

Synonym: none.

Syntax: "TMP=" <directory>.

Arguments: <directory>: Directory to be used for temporary files.

Default: none.

Description: If a temporary file has to be created, normally the ANSI function tmpnam() is used. This library function stores the temporary files created in the directory specified by this environment variable. If the variable is empty or does not exist, the current directory is used. Check this variable if you get an error message “Cannot create temporary file”.

NOTE This is an environment variable on system level (global environment variable) It CANNOT be specified in a default environment file (DEFAULT.ENV/.hidefaults).

Example: TMP=C:\TEMP

See also: Section ‘The Current Directory’

USERNAME

USERNAME: User Name in Object File

Tools: Compiler, Assembler, SmartLinker, Librarian

Synonym: None

Syntax: "USERNAME=" <user>.

Arguments: <user>: Name of user.

Default: None

Description: Each absolute file contains an entry identifying the user who created the file. This information may be retrieved from the absolute files using the decoder.

Example: USERNAME=PowerUser

See also: Environment variable COPYRIGHT

Environment variable INCLUDETIME

4

Files

Input FilesOutput Files

Input Files

Parameter File

The linker takes any file as input, it does not require the file name to have a special extension. However, we suggest that all your parameter file names have extension .prm.. Parameter file will be searched first in the project directory and then in the directories enumerated in GENPATH. The parameter file must be a strict ASCII text file.

Object File

The list of files to be linked is specified in the link parameter file entry NAMES. Additional object files can be specified with the option -Add.

The linker looks for the object files first in the project directory, then in the directories enumerated in OBJPATH and finally in the directories enumerated in GENPATH. The binary files must be valid HIWARE, ELF\DWARF 1.1 or 2.0 objects, absolute or library files.

Output Files

Absolute Files

After successful linking session, the SmartLinker generates an absolute file containing the target code as well as some debugging information. This file is written to the directory given in the environment variable ABSPATH. If that variable contains more than one path, the absolute file is written in the first directory given; if this variable is

not set at all, the absolute file is written in the directory the parameter file was found. Absolute files always get the extension .abs.

Motorola S Files

After successful linking session and if the option -B is present, the SmartLinker generates a Motorola S record file, which can be burnt into an EPROM. This file contains information stored in all the READ_ONLY sections in the application. The extension for the generated Motorola S record file depends on the setting from the variable SRECORD.

- If SRECORD = S1, the Motorola S record file gets the extension `.s1`.
- If SRECORD = S2, the Motorola S record file gets the extension `.s2`.
- If SRECORD = S3, the Motorola S record file gets the extension `.s3`.
- If SRECORD is not set, the Motorola S record file gets the extension `.sx`.

This file is written to the directory given in the environment variable ABSPATH. If that variable contains more than one path, the S record file is written in the first directory given; if this variable is not set at all, the S record file is written in the directory the parameter file was found.

Map Files

After successful linking session, the SmartLinker generates a map file containing information about the link process. This file is written to the directory given in the environment variable TEXTPATH. If that variable contains more than one path, the map file is written in the first directory given; if this variable is not set at all, the map file is written in the directory the parameter file was found. map files always get the extension .map.

Dependency Information

The linker provides useful dependency information in the map file generated. Basically the dependency information shows which object are used by an object (function, variable, ...).

The dependency information in the linker map file is based on fixups/relocations. That is if an object references another object by a relocation, this object is added to the dependency list.

Example:

```
int bar;
void foo(void) {
  bar = 0;
}
```

In the above example, in foo the compiler has generated a fixup/relocation to the object bar, so the linker knows that foo uses bar. For the next example, foo will reference foo itself, because in foo there is a fixup to foo as well:

```
void foo(void) {
  foo();
}
```

Now it could be that the compiler will do a common code optimization, that is if the compiler tries to collect some common code in a function so that the code size can be reduced. Note that you can switch off this compiler common code optimization.

Example:

```
void foo(void) {
  if (bar == 3) bar = 0;
  ...
  if (bar == 3) bar = 0;
}
```

In the above example, the compiler could optimize this to

```
int foo(void) {
  bsr foo:Label:
  ...
  foo_Label:
    if (bar == 3) bar = 0;
    return;
}
```

Here the compiler will generate a local branch inside foo to a local subroutine. This produces a relocation/fixup into foo, that is for the linker foo references itself.

Error Listing File

If the SmartLinker detects any errors, it does not create an absolute file but an error listing file. This file is generated in the directory the source file was found (also see Environment, Environment Variable ERRORFILE).

If the Linker window is open, it displays the full path of all binary files read. In case of error, the position and file name where the error occurs is displayed in the linker window.

If the SmartLinker is started from WinEdit (with '%f' given on the command line) or Codewright (with '%b%e' given on the command line), this error file is not produced. Instead it writes the error messages in a special format in a file called EDOUT using the Microsoft format by default. Use WinEdit's 'Next Error' or Codewright's 'Find Next Error' command to see both error positions and the error messages.

Interactive Mode (SmartLinker window open)

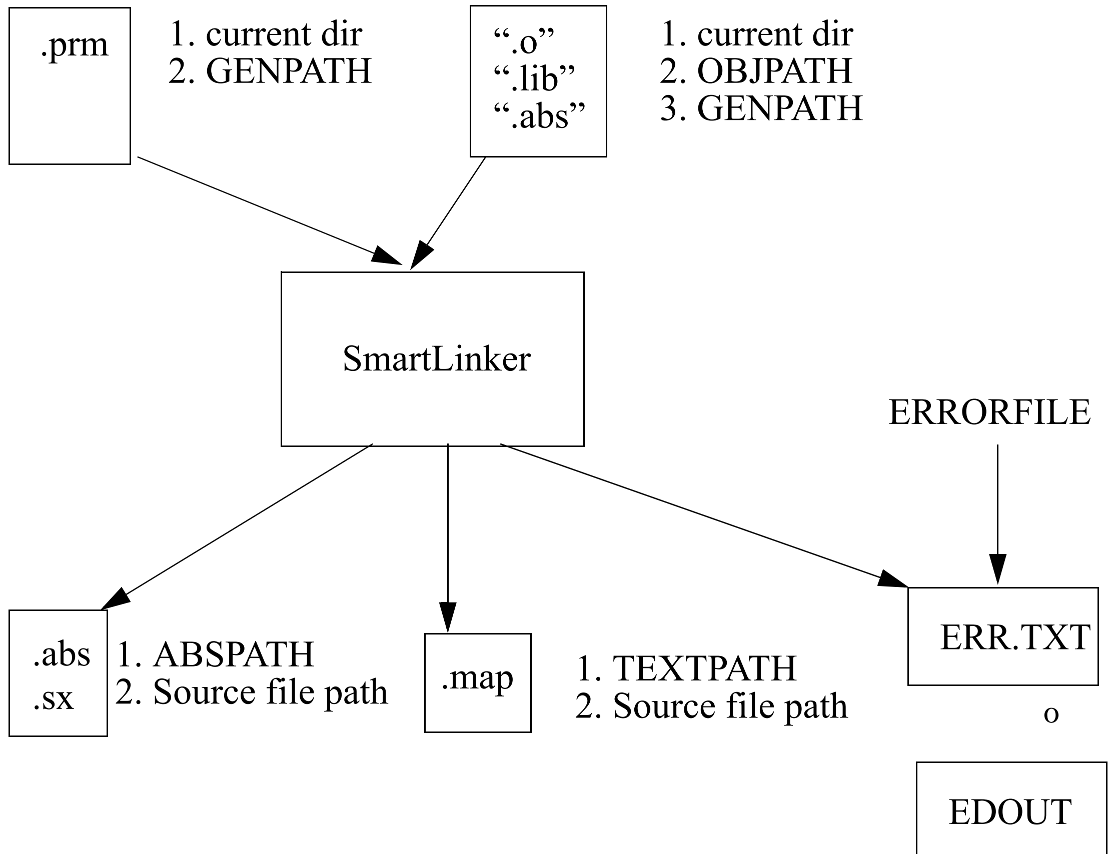
If ERRORFILE is set, the SmartLinker creates a message file named as specified in this environment variable.

If ERRORFILE is not set, a default file named ERR.TXT is generated in the current directory.

Batch Mode (SmartLinker window not open)

If ERRORFILE is set, the SmartLinker creates a message file named as specified in this environment variable.

If ERRORFILE is not set, a default file named EDOUT is generated in the current directory.


.prm
1. current dir
2. GENPATH
“.o”
“.lib”
“.abs”
1. current dir
2. OBJPATH
3. GENPATH
SmartLinker
ERRORFILE
.abs
.sx
1. ABSPATH
2. Source file path
.map
1. TEXTPATH
2. Source file path
ERR.TXT
o
EDOUT

5

SmartLinker Options

The SmartLinker offers a number of options that you can use to control the SmartLinker's operation. Options are composed of a minus/dash ('-') followed by one or more letters or digits. Anything not starting with a dash/minus is supposed to be the name of a parameter file to be linked. SmartLinker options may be specified on the command line or in the LINKOPTIONS variable. Typically, each linker option is specified only once per linking session.

Command line options are not case sensitive, for example, "–W1" is the same as "–w1".

LINKOPTIONS

If this environment variable is set, the linker appends its contents to its command line each time a new file is linked. It can be used to globally specify certain options that should always be set, so you do not have to specify them each time a file is linked.

SmartLinker Option Details

The remainder of this section is devoted to describing each of the SmartLinker options available for the SmartLinker. Table 5.1 lists the options in alphabetical order and each of the options is divided into several sections.

-Add

Table 5.1 SmartLinker Option Details

Topic	Description
Group	Specifies what sort of influence this option has.
Syntax	Specifies the syntax of the option in a EBNF format.
Arguments	Describes and lists optional and required arguments for the option.
Default	Shows the default setting for the option.
Description	Provides a detailed description of the option and how to use it.
Example	Gives an example of usage, and effects of the option where possible. SmartLinker settings, source code and/or SmartLinker prm files are displayed where applicable. The examples shows an entry in the default.env for PC or in the .hidefaults for UNIX.
See also	Names related options.

-Add: Additional Object/Library File

Group: INPUT

Syntax: "-Add" <FileList>.

Arguments: <FileList>: Names of an additional object files or libraries.

Default: none.

Description: With the option -Add, additional files can be added to a project without modifying the link parameter file.
If all binary files should be specified by the command line option -add, then an empty NAMES block (just NAMES END) must be present in the link parameter file.
Object files added with this option are linked before the object files specified in the NAMES block.

To specify more than one file either use several options -Add:

linker.exe demo.prm -addFileA.o -addFileB.o

or use braces to bind the list to the option -add:

linker.exe demo.prm -add(FileA.o FileB.o)

To add a file which name contain spaces, use braces together with double quotes:

linker.exe demo.prm -add(“File A.o” “File B.o”)

NOTE To switch off smart linking for the additional object file, use a + sign immediately behind the filename.

Example: *linker.exe fibo.prm -addfibo1.o -addfibo2.o*
In this example, the additional object files fibo1.o and fibo2.o are linked with the fibo application.

See also: command NAMES.

-AllocFirst,-AllocNext,-AllocChange

-Alloc: Allocation over segment boundaries (ELF)

Group: OPTIMIZATION

Syntax: "-Alloc" ("First" | "Next" | "Change")

Arguments: “First”: use first free location

“Next”: always use next segment

“Change”: check when segment changes only

Default: -AllocNext.

Description: The linker supports to allocate objects from one ELF section into different segments. The allocation strategy controls where space for the next object is allocated as soon as the first segment is full.

In the AllocNext strategy, the linker does always take the next segment as soon as the current segment is full. Holes generated during this process are not used later. With this strategy, the allocation order corresponds to the definition order in the object files. Objects defined first in a source file are allocated before later defined objects.

In the AllocFirst strategy, the linker checks for every object, if there is a previously only partially used segment, into which the current object does fit. This strategy does not maintain the definition order.

In the AllocChange strategy, the linker checks as soon as a object does no longer fit into the current segment, if there is a previously only partially used segment, into which the current object does fit. This strategy does not maintain the definition order, but it does however use fewer different ranges than the AllocFirst case.

NOTE This option has no effect in the HIWARE format. In the HIWARE format, the linker does always use the "-AllocNext" strategy. However, the linker does not maintain the allocation order for small variables.

NOTE This option has no effect if sections are not split into several segments. Then all strategies behave identically.

NOTE Some compilers do optimization in the assumption that the definition order is maintained in the memory. But for such code, no splitting up into several segment is allowed anyway, so this optimization does not cause new problems.

Example: Objects: AAAA BB CCC D EEE FFFFF
Segments: "---" "-------" "------------"
AllocNext: "---" "AAAABB-" "CCCDEEEFFFFF"
AllocChange:"CCC" "AAAABBD" "EEEFFFFF----"
AllocFirst: "BBD" "AAAACCC" "EEEFFFFF----"

In this example, the objects A (size 4), B (size 2), ... F (size 5) should be allocated into 3 segments of size 3, 7 and 12 bytes. Because the object A does not fit into the first segment, the AllocNext strategy does not use this space at all. The two other strategies are filling this space later. The order of the objects is only maintained by the AllocNext case.

See also: None.

-AsROMLib

-AsROMLib: Link as ROM Library

Group: OUTPUT

Syntax: "-AsROMLib".

Arguments: <FileList>: Names of an additional object files or libraries.

Default: none.

Description: With the option -AsROMLib set, the application is linked as a ROM library. This option has the same effect as specifying AS_ROM_LIB in the linker parameter file.

Example: *linker.exe myROMlib.prm -AsROMLib*

See also: AS_ROM_LIB.

-B

-B: Generate S-Record file

Group:	OUTPUT
Syntax:	"-B".
Arguments:	none.
Default:	Disabled.
Description:	This option specifies that in addition to an absolute file, also a srecord file should be generated. The name of the srecord file is the same as the name of the abs file, except that the extension “SX” is used. The default.env variable “SRECORD” may specify an alternative extension.
Example:	*LINKOPTIONS=-B*
See also:	none.

-CAllocUnusedOverlap

-CAllocUnusedOverlap: Allocate not referenced overlap variables (HIWARE)

Group:	OPTIMIZATION
Syntax:	"-CAllocUnusedOverlap”.
Arguments:	none.
Default:	none.

Description: When Smart Linking is switched off, not referenced, but defined overlap variables are still not allocated by default. Such variables do not belong to a specific function. Therefore they cannot be allocated overlapped with other variables.

Note that this option does only change the behavior of variables in the special _OVERLAP segment. This segment is only used for the purpose of allocating parameters and local variables for processors, which do not have a stack. Not allocating a non referenced overlap variable therefore is similar to not allocating a variable on the stack for other processors. If you use this stack analogy, then allocating such variables this way corresponds to allocate not referenced stack variables in global memory.

This option is provided to make it possible to allocate all defined objects. It is not recommended to use this option.

Example: *LINKOPTIONS=-CAllocUnusedOverlap*

See also: Ovelapping Locals

segment _OVERLAP

-Ci

-Ci: Link Case Insensitive

Group: INPUT

Syntax: "-Ci".

Arguments: none.

Default: none.

Description: With this option, the linker compares all object names case insensitive.
The main purpose for this option is to support case insensitive linking of assembly modules. But because all identifiers are linked case insensitive, this also affects C or C++ modules.
This option might cause sever problems with the name mangling of C++, therefore it should not be used with C++.
This option does only affect the comparison of names of linked objects. Section names or the parsing of the link parameter file are not affected. They remain case sensitive.

Example: *void Fun(void);*
void main(void) {
  fun(); /* with -ci this call is resolved to Fun */
}
The linker will match the fun and Fun identifiers at link time. However, for the compiler these are still two separate objects and therefore the code above issues a “implicit parameter declaration” warning.

See also: none.

-Cocc

-Cocc: Optimize Common Code (ELF)

Group: OPTIMIZATION

Syntax: "-Cocc"[“=”[“D”] [“C”]].

Arguments: “D”: optimize Data (constants and strings).
“C”: optimize Code

Default: none.

Description: This option defines the default if constants and code should be optimized. The commands DO_OVERLAP_CONSTS and DO_NOT_OVERLAP_CONSTS take precedence over the option.

Example: *printf(“Hello World\n“); printf(“\n”);*
With -Cocc, the string “\n” is allocated inside of the string “Hello World\n”.

See also: Command DO_OVERLAP_CONSTS

-CRam

-CRam: Allocate non specified const segments in RAM (ELF)

Group: OPTIMIZATION

Syntax: "-CRam".

Arguments: none.

Default: none.

Description: With this option, constant data segments not explicitly allocated in a READ_ONLY segment are allocated in the default READ_WRITE segment.

This was the default for old versions of the linker, so this option provides a compatible behavior with old linker versions.

Example: When C source files are compiled with -CC, the constants are put into the ROM_VAR segment. If the ROM_VAR segment is not mentioned in the prm file, then without this option, these constants are allocated in DEFAULT_ROM. With this option they are allocated in DEFAULT_RAM.

See also: none.

-Dist

-Dist: Enable distribution optimization (ELF)

Group: OPTIMIZATIONS

Syntax: "-Dist".

Arguments: none.

Default: none .

Description: With this option the linker optimizer is enabled. Instead of link the linker generates a distribution file which contains a optimized distribution.

See also: Automatic Distribution of Paged Functions

-DistFile

-DistFile: Specify distribution file name (ELF)

Group: OPTIMIZATIONS

Syntax: "-DistFile"<file name>.

Arguments: <file name>: Name of the distribution file.

Default: distr.inc .

Description: When this option is enabled, it‘s possible to specify the name of the distribution file. There are listed all distributed functions and how the compiler has to reallocate them.

Example: *LINKOPTIONS=-DistFileMyFile*

See also: Automatic Distribution of Paged Functions

-DistInfo

-DistInfo: Generate distribution information file (ELF)

Group: OPTIMIZATIONS

Syntax: "-DistInfo"<file name>.

Arguments: <file name>: Name of the information file.

Default: distr.txt .

Description: When this option is enabled, the optimizer generates a distribution information file with a list of all sections and their functions. To the functions are several informations available like: old size, optimized size and new calling convention.

Example: *LINKOPTIONS=-DistInfoMyInfoFile*

See also: Automatic Distribution of Paged Functions

-DistOpti

-DistOpti: Choose optimizing method (ELF)

Group: OPTIMIZATIONS

Syntax: "-DistOpti" ("FillBanks" | "CodeSize")

Arguments: “FillBanks”: Priority is to fill the banks

“CodeSize”: Priority is to minimize the code size

Default: -DistOptiFillBanks.

Description: When this option is enabled, it‘s possible to choose the optimizing method. With the argument “FillBanks” the priority for the linker is the minimization of the free space in every bank. This method has the disadvantage that less functions have a near calling convention. If the code size has to be minimized and the free space which remains in the banks is no problem so it is recommendable to use the argument “CodeSize”.

Example: *LINKOPTIONS=-DistOptiFillBanks*

See also: Automatic Distribution of Paged Functions

-DistSeg

-DistSeg: Specify distribution segment name (ELF)

Group: OPTIMIZATIONS

Syntax: "-DistSeg"<segment name>.

Arguments: <segment name>: Name of the distribution segment.

Default: DISTRIBUTE .

Description: When this option is enabled, it‘s possible to specify the name of the distribution segment.

Example: *LINKOPTIONS=-DistSegMyDistributionSegment*

See also: Automatic Distribution of Paged Functions

-E

-E: Define Application Entry Point (ELF)

Group: INPUT

Syntax: "-E=" <FunctionName>.

Arguments: <FunctionName>: Name of the function which is considered to be the entry point in the application.

Default: none.

Description: This option specifies the name of the application entry point.

The symbol specified must be a externally visible (not defined as static in an ANSI C source file or XREFed in an assembly source file).

Example: LINKOPTIONS=-E=entry
This is the same as using the command:

NIT entry

in the prm file

See also: Command INIT

-Env

-Env: Set Environment Variable

Group: HOST

Syntax: "-Env" <Environment Variable> "=" <Variable Setting>.

Arguments: <Environment Variable>: Environment variable to be set
<Variable Setting>: Setting of the environment variable

Default: none.

Description: This option sets an environment variable.

Example: “-EnvOBJPATH=\sources\obj”

This is the same as:
OBJPATH=\sources\obj
in the default.env

See also: none.

-FA, -FE, -FH -F6

-FA, -FE, -FH -F6: Object File Format

Group: INPUT.

Syntax: "-F" (“A” | “E” | “H” | “6”).

Arguments: none.

Default: “-FA”

Description: The linker is able to link different object file formats.

This option defines which object file format should be used.

With “-FA”, the linker determines the object file format automatically. With “-F2”, this automatism can be overridden and only ELF files are correctly recognized. With “-FH” only HIWARE files are known. With “-F6” set, the linker produces a V2.6 HIWARE absolute file.

NOTE It is not possible to build an application consisting of some HIWARE and some ELF files. Either all files are in the ELF format or all files are in the HIWARE format.
The format of the generated absolute file is the same as the format of the object files. ELF objects files generate a ELF absolute file and HIWARE object files generate a HIWARE absolute file.

See also: none.

-H

-H: Prints the List of All Available Options

Group: OUTPUT.

Syntax: "-H".

Arguments: none.

Default: none.

Description: Prints the list of all options of the SmartLinker.

The options are sorted by the Group. Options in the same group, are sorted alphabetically.

See also: none.

-L

-L: Add a path to the search path (ELF)

Group: INPUT

Syntax: "-L" <Directory>.

Arguments: <Directory>: Name of an additional search directory for object files.

Default: none.

Description: With this option, the ELF part of this linker searches object files first in all paths given with this option. Then the usual environment variables are considered.

Example: *LINKOPTIONS=-Lc:\metrowerks\obj*

See also: Environment Variable OBJPATH

-Lic

-Lic: Print license information

Group: Various.

Syntax: "-Lic"

Arguments: none.

Default: none.

Description: This options shows the current state of the license information.

When no full license is available, the SmartLinker runs in demo mode.

In demo mode, the size of the applications which can be linked is limited

Example: none.

See also: Option -Lica

-LicA

-LicA: License Information about every Feature in Directory

Group: Various

Syntax: "-LicA".

Arguments: none.

Default: none.

Defines: none.

Description: The -LicA option prints the license information of every tool or dll in the directory were the executable is (e.g. if tool or feature is a demo version or a full version). Because the option has to analyze every single file in the directory, this takes a long time.

Example: none.

See also: Option -Lic

-M

-M: Generate Map File

Group: OUTPUT

Syntax: "-M"

Arguments: None.

Default: none.

Description: This option force the generation of a map file after a successful linking session.

Example: *LINKOPTIONS=-M*

This is the same as using the command:

MAPFILE ALL

in the prm file

See also: Command MAPFILE

-N

-N: Display Notify Box

Group: MESSAGE

Syntax: "-N".

Arguments: none.

Default: none.

Description: Makes the SmartLinker display an alert box if there was an error during linking. This is useful when running a makefile since the linker waits for the user to acknowledge the message, thus suspending makefile processing. (The 'N' stands for “Notify”.)

This feature is useful for halting and aborting a build using the Make Utility.

Example: *LINKOPTIONS=-N*

If during linking an error occurs, an error dialog box will be opened.

See also: none.

-NoBeep

-NoBeep: No Beep in Case of an Error

Group: MESSAGE

Syntax: "-NoBeep".

Arguments: none.

Default: none.

Description: Normally there is a ‘beep’ notification at the end of processing if there was an error. To have a silent error behavior, this ‘beep’ may be switched of using this option.

Example: none.

See also: none.

-NoEnv

-NoEnv: Do not use Environment

Group: Startup. (This option can not be specified interactively)

Syntax: "-NoEnv".

Arguments: none.

Default: none.

Description: This option can only be specified at the command line while starting the application. It can not be specified in any other circumstances, including the default.env file, the command line or whatever.

When this option is given, the application does not use any environment (default.env, project.ini or tips file).

Example: *linker.exe -NoEnv*

See also: Section Environment

-OCopy

-OCopy: Optimize Copy Down (ELF)

Group: OPTIMIZATION

Syntax: "-OCopy" (“On” | “Off”).

Arguments: On: Do the optimization.

Off: Optimization disabled

Default: -OCopyOn.

Description: This optimization changes the copy down structure to use as few space as possible.

The optimization does assume that the application does perform both the zero out and the copy down step of the global initialization. If a value is set to zero by the zero out, then zero values are removed from the copy down information. The resulting initialization is not changed by this optimization if the default startup code is used.

This switch does only have an effect in the ELF Format. The optimizations done in the HIWARE format cannot be switched off.

Example: *LINKOPTIONS=-OCopyOn*

See also: Program Startup

-O

-O: Define Absolute File Name

Group: OUTPUT

Syntax: "-O" <FileName>.

Arguments: <fileName>: Name of the absolute file which must be generated by the linking session.

Default: none.

Description: This option defines the name of the ABS file which must be generated. If you are using the Linker with CodeWarrior, then this option is automatically added to the command line passed to the linker. You can see this if you enable ‘Display generated command lines in message window’ in the Linker preference panel in CodeWarrior.
No extension is added automatically. For the option “-otest”, a file named “test” is generated. To get the usual file extension “abs”, use “-otest.abs”.

Example: *LINKOPTIONS=-Otest.abs*
This is the same as using the command:
LINK test.abs
in the prm file.

See also: Command LINK

-Prod

-Prod: specify project file at startup (PC)

Group: none. (this option can not be specified interactively)

Syntax: "-Prod=”<file>.

Arguments: <file>: name of a project or project directory

Default: none.

Description: This option can only be specified at the command line while starting the linker. It can not be specified in any other circumstances, including the default.env file, the command line or whatever.

When this option is given, the linker opens the file as configuration file. When the file name does only contain a directory, the default name project.ini is appended. When the loading fails, a message box appears.

Example: *linker.exe -prod=project.ini*

See also: none.

-S

-S: Do not generate DWARF Information (ELF)

Group: OUTPUT

Syntax: "-S"

Arguments: None.

Default: none.

Description: This option disables the generation of DWARF sections in the absolute file. This allow you to save some memory on your PC.

NOTE If the absolute file does not contain any DWARF information, you will not be able to debug it any more symbolically.

Example: *LINKOPTIONS=-S*

See also: None

-SFixups

-SFixups: Creating Fixups (ELF)

Group: OUTPUT

Syntax: "-SFixups".

Arguments: none.

Default: none.

Description: Usually, absolute files do not contain any fixups because all fixups are evaluated at link time. But with fixups, the decoder might symbolically decode the content in absolute files, which is not possible without fixups. Some debuggers do not load absolute files which contain fixups because they assume that these fixups are not yet evaluated. But the fixups inserted with this option are actually already handled by this linker.

This option is contained mainly because of compatibility with previous versions of the linker.

Example: *LINKOPTIONS=-SFixups*

See also: none.

-StatF

-StatF: Specify the name of statistic file

Group: OUTPUT

Syntax: "-StatF="<fileName>.

Arguments: <fileName>: name for the file to be written

Default: none.

Description: With this option set, the linker generates a statistic file. In this file, each allocated object is reported with it’s attributes. Every attribute is separated by a TAB character, so it can be easily imported into a spreadsheet/database program for further processing

Example: *LINKOPTIONS=-StatF*

See also: none.

-V: Prints the SmartLinker Version

Syntax: "-V".

Arguments: none.

Default: none.

Description: Prints the SmartLinker version and the project directory

This option is useful to determine the project directory of the SmartLinker.

Example: -V produces the following list:

Directory: \software\sources\asm

SmartLinker, V5.0.4, Date Apr 20 1997

See also: none.

-View

-View: Application Standard Occurrence (PC)

Group: HOST

Syntax: "-View" <kind>.

Arguments: <kind> is one of:

“Window”: Application window has default window size

“Min”: Application window is minimized

“Max”: Application window is maximized

“Hidden”: Application window is not visible (only if arguments)

Default: Application started with arguments: Minimized.

Application started without arguments: Window.

Description: Normally the application (e.g. linker, compiler, ...) is started as normal window if no arguments are given. If the application is started with arguments (e.g. from the maker to compile/link a file) then the application is running minimized to allow batch processing. However, with this option the behavior may be specified. Using -ViewWindow the application is visible with its normal window. Using -ViewMin the application is visible iconified (in the task bar). Using -ViewMax the application is visible maximized (filling the whole screen). Using -ViewHidden the application processes arguments (e.g. files to be compiled/linked) completely invisible in the back ground (no window/icon in the taskbar visible). However e.g. if you are using the option -N a dialog box is still possible.

Example: -ViewHidden fibo.prm

See also: none.

-W1

-W1: No Information Messages

Group: MESSAGE

Syntax: "-W1".

Arguments: none.

Default: none.

Description: Inhibits the Linker to print INFORMATION messages, only WARNING and ERROR messages are emitted.

Example: *LINKOPTIONS=-W1*

See also: None

-W2

-W2: No Information and Warning Messages

Group:	MESSAGE
Syntax:	"-W2".
Arguments:	none.
Default:	none.
Description:	Suppresses all messages of type INFORMATION and WARNING, only ERRORs are printed.
Example:	LINKOPTIONS=-W2
See also:	None

-WErrFile

-WErrFile: Create "err.log" Error File

Group: MESSAGE

Syntax: "-WErrFile" ("On" | "Off").

Arguments: none.

Default: err.log is created/deleted.

Description: The error feedback from the compiler to called tools is now done with a return code. In 16 bit windows environments, this was not possible, so in the error case a file “err.log” with the numbers of errors written into was used to signal an error. To state no error, the file “err.log” was deleted. Using UNIX or WIN32, there is now a return code available, so this file is no longer needed when only UNIX / WIN32 applications are involved. To use a 16 bit maker with this tool, the error file must be created in order to signal any error.

Example:

-WErrFileOn

err.log is created/deleted when the application is finished.

-WErrFileOff

existing err.log is not modified.

See also: Option -WStdout

Option -WOutFile

-Wmsg8x3

-Wmsg8x3: Cut file names in Microsoft format to 8.3 (PC)

Group: MESSAGE

Syntax: "-Wmsg8x3".

Arguments: none.

Default: none.

Description: Some editors (e.g. early versions of WinEdit) are expecting the file name in the Microsoft message format in a strict 8.3 format, that means the file name can have at most 8 characters with not more than a 3 characters extension. Using Win95 or WinNT longer file names are possible. With this option the file name in the Microsoft message is truncated to the 8.3 format.

Example: *x:\mysourcefile.prm(3): INFORMATION C2901: Unrolling loop*

With the option -Wmsg8x3 set, the above message will be

x:\mysource.c(3): INFORMATION C2901: Unrolling loop

See also: Option -WmsgFi

Option -WmsgFb

Option -WmsgFoi

Option -WMsgFob

Option -WmsgFonP

-WmsgCE

-WmsgCE: RGB color for error messages

Group: MESSAGE

Scope: Function

Syntax: "-WmsgCE" <RGB>.

Arguments: <RGB>: 24bit RGB (red green blue) value.

Default: -WmsgCE16711680 (rFF g00 b00, red)

Defines: none.

Description: With this options it is possible to change the error message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and has to be specified in decimal.

Example: -WmsgCE255 changes the error messages to blue.

See also: none.

-WmsgCF

-WmsgCF: RGB color for fatal messages

Group:	MESSAGE
Scope:	Function
Syntax:	"-WmsgCF" <RGB>.
Arguments:	<RGB>: 24bit RGB (red green blue) value.
Default:	-WmsgCF8388608 (r80 g00 b00, dark red)
Defines:	none.
Description:	With this options it is possible to change the fatal message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and has to be specified in decimal.
Example:	-WmsgCF255 changes the fatal messages to blue.
See also:	none.

-WmsgCI

-WmsgCI: RGB color for information messages

Group:	MESSAGE
Scope:	Function
Syntax:	"-WmsgCI" <RGB>.
Arguments:	<RGB>: 24bit RGB (red green blue) value.
Default:	-WmsgCI32768 (r00 g80 b00, green)
Defines:	none.
Description:	With this options it is possible to change the information message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and has to be specified in decimal.
Example:	-WmsgCI255 changes the information messages to blue.
See also:	none.

-WmsgCU

-WmsgCU: RGB color for user messages

Group:	MESSAGE
Scope:	Function
Syntax:	"-WmsgCU" <RGB>.
Arguments:	<RGB>: 24bit RGB (red green blue) value.
Default:	-WmsgCU0 (r00 g00 b00, black)
Defines:	none.
Description:	With this options it is possible to change the user message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and has to be specified in decimal.
Example:	-WmsgCU255 changes the user messages to blue.
See also:	none.

-WmsgCW

-WmsgCW: RGB color for warning messages

Group:	MESSAGE
Scope:	Function
Syntax:	"-WmsgCW" <RGB>.
Arguments:	<RGB>: 24bit RGB (red green blue) value.
Default:	-WmsgCW255 (r00 g00 bFF, blue)
Defines:	none.
Description:	With this options it is possible to change the warning message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and has to be specified in decimal.
Example:	-WmsgCW0 changes the warning messages to black.
See also:	none.

-WmsgFb (-WmsgFbv, -WmsgFbm)

-WmsgFb: Set message file format for batch mode

Group: MESSAGE

Syntax: "-WmsgFb" ["v" | "m"].

Arguments: "v": Verbose format.

"m": Microsoft format.

Default: -WmsgFbm

Description: The SmartLinker can be started with additional arguments (for example, files to be linked together with SmartLinker options). If the SmartLinker has been started with arguments (for example, from the Make Tool or with the '%f' argument from WinEdit), the SmartLinker links the files in a batch mode, that is no SmartLinker window is visible and the SmartLinker terminates after job completion.

If the linker is in batch mode the linker messages are written to a file instead to the screen. This file only contains the linker messages (see examples below).

By default, the SmartLinker uses a Microsoft message format to write the SmartLinker messages (errors, warnings, information messages) if the linker is in batch mode.

With this option, the default format may be changed from the Microsoft format (only line information) to a more verbose error format with line, column and source information

Example:

```
LINK fibo2.abs
NAMES fibo.o start12s.o ansis.lib END
PLACEMENT
  .text INTO READ_ONLY 0x810 TO 0xAFF;
  .data INTO READ_WRITE 0x800 TO 0x80F
END
```

By default, the SmartLinker generates following error output in the SmartLinker window if it is running in batch mode:

X:\fibo2.prm(7): ERROR L1004: ; expected

Setting the format to verbose, more information is stored in the file:

LINKOPTIONS=-WmsgFbv

>> in "X:\fibo2.prm", line 7, col 0, pos 159

.data INTO READ_WRITE 0x800 TO 0x80F

END

^

ERROR L1004: ; expected

See also: Option -WmsgFi

-WmsgFi (-WmsgFiv, -WmsgFim)

-WmsgFi: Set message file format for Interactive mode

Group: MESSAGE

Syntax: "-WmsgFi" ["v" | "m"].

Arguments: "v": Verbose format.

"m": Microsoft format.

Default: -WmsgFiv

Description: If the SmartLinker is started without additional arguments (e.g. files to be linked together with SmartLinker options), the SmartLinker is in the interactive mode (that is, a window is visible).

By default, the SmartLinker uses the verbose error file format to write the SmartLinker messages (errors, warnings, information messages).

With this option, the default format may be changed from the verbose format (with source, line and column information) to the Microsoft format (only line information).

With this option, the default format may be changed from the Microsoft format (only line information) to a more verbose error format with line, column and source information

NOTE	Using the Microsoft format may speed up the compilation, because the SmartLinker has to write less information to the screen.

Example: *LINK fibo2.abs*

NAMES fibo.o start12s.o ansis.lib END

PLACEMENT

.text INTO READ_ONLY 0x810 TO 0xAFF;

.data INTO READ_WRITE 0x800 TO 0x80F

END

By default, the SmartLinker following error output in the SmartLinker window if it is running in interactive mode

>> in "X:\fibo2.prm", line 7, col 0, pos 159

.data INTO READ_WRITE 0x800 TO 0x80F

END

^

ERROR L1004: ; expected

Setting the format to Microsoft, less information is displayed:

LINKOPTIONS=-WmsgFim

X:\fibo2.prm(7): ERROR L1004: ; expected

See also: Option -WmsgFb

-WmsgFob

-WmsgFob: Message format for Batch Mode

Group: MESSAGE

Syntax: "-WmsgFob"<string>.

Arguments: <string>: format string (see below).

Default: -WmsgFob"%”%f%e%”(%l): %K %d: %m\n"

Description: With this option it is possible modify the default message format in batch mode. Following formats are supported (supposed that the source file is x:\metrowerks\sourcefile.prmx)

Format	Description	Example
%s	Source Extract	
%p	Path	x:\metrowerks\
%f	Path and name	x:\metrowerks\sourcefile
%n	File name	sourcefile
%e	Extension	.prmx
%N	File (8 chars)	sourcefi
%E	Extension (3 chars)	.prm
%l	Line	3
%c	Column	47
%o	Pos	1000
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	L1051
%m	Message	text
%”	” if full name contains a space	”
%’	’ if full name contains a space	
%%	Percent	%
\n	New line	

Example: *LINKOPTIONS=-WmsgFob”%’%f%e%’(%l): %k %d: %m\n”*

produces a message in following format:

x:\metrowerks\sourcefile.prmx(3): error L1000: LINK not found

See also: Environment variable ERRORFILE
Option -WmsgFb
Option -WmsgFi
Option -WmsgFonp
Option -WmsgFonf
Option -WmsgFoi

-WmsgFoi

-WmsgFoi: Message Format for Interactive Mode

Group: MESSAGE

Syntax: "-WmsgFoi"<string>.

Arguments: <string>: format string (see below).

Default: -WmsgFoi"\n>> in \"%"%f%e%"\", line %l, col %c, pos %o\n%s\n%K %d: %m\n"

Description: With this option it is possible modify the default message format in interactive mode. Following formats are supported (supposed that the source file is x:\metrowerks\sourcefile.prmx):

Format	Description	Example
%s	Source Extract	
%p	Path	x:\metrowerks\
%f	Path and name	x:\metrowerks\sourcefile
%n	File name	sourcefile
%e	Extension	.prmx
%N	File (8 chars)	sourcefi
%E	Extension (3 chars)	.prm

%l	Line	3
%c	Column	47
%o	Pos	1234
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	L1000
%m	Message	text
%”	” if full name contains a space	”
%’	’ if full name contains a space	
%%	Percent	%
\n	New line	

Example: *LINKOPTIONS=-WmsgFoi”%f%e(%l): %k %d: %m\n”*

produces a message in following format:

x:\metrowerks\sourcefile.prmx(3): error L1000: LINK not found

See also: Environment variable ERRORFILE

Option -WmsgFb

Option -WmsgFi

Option -WmsgFonp

Option -WmsgFonf

Option -WmsgFob

-WmsgFonf

-WmsgFonf: Message Format for no File Information

Group: MESSAGE

Syntax: "-WmsgFonf"<string>.

Arguments: <string>: format string (see below).

Default: -WmsgFonf"%K %d: %m\n"

Description: Sometimes there is no file information available for a message (for example, if a message not related to a specific file). Then this message format string is used. Following formats are supported:

Format	Description	Example
-		
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	L10324
%m	Message	text
%%	Percent	%
\n	New line	

Example: *LINKOPTIONS=-WmsgFonf"%k %d: %m\n"*

produces a message in following format:

information L10324: Linking successful

See also: Environment variable ERRORFILE

Option -WmsgFb

Option -WmsgFi

Option -WmsgFonp

Option -WmsgFoi

Option -WmsgFob

-WmsgFonp

-WmsgFonp: Message Format for no Position Information

Group: MESSAGE

Syntax: "-WmsgFonp"<string>.

Arguments: <string>: format string (see below).

Default: -WmsgFonp"%”%f%e%”: %K %d: %m\n"

Description: Sometimes there is no position information available for a message (e.g. if a message not related to a certain position). Then this message format string is used. Following formats are supported (supposed that the source file is x:\metrowerks\sourcefile.prmx)

Format	Description	Example
%p	Path	x:\metrowerks\
%f	Path and name	x:\metrowerks\sourcefile
%n	File name	sourcefile
%e	Extension	.prmx
%N	File (8 chars)	sourcefi
%E	Extension (3 chars)	.prm
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	L10324
%m	Message	text
%”	” if full name contains a space	”
%’	’ if full name contains a space	

%%	Percent	%
\n	New line	

Example: *LINKOPTIONS=-WmsgFonf"%k %d: %m\n"*

produces a message in following format:

information L10324: Linking successful

See also: Environment variable ERRORFILE

Option -WmsgFb

Option -WmsgFi

Option -WmsgFonf

Option -WmsgFoi

Option -WmsgFonfob

-WmsgNe

-WmsgNe: Number of Error Messages

Group: MESSAGE

Syntax: "-WmsgNe" <number>.

Arguments: <number>: Maximum number of error messages.

Default: 50

Description: With this option the amount of error messages can be set until the SmartLinker stops the current linking session. Note that subsequent error messages which depends on a previous one may be confusing.

Example: LINKOPTIONS=-WmsgNe2

The SmartLinker stops compilation after two error messages.

See also: Option -WmsgNi

Option -WmsgNw

-WmsgNi

-WmsgNi: Number of Information Messages

Group: MESSAGE

Syntax: "-WmsgNi" <number>.

Arguments: <number>: Maximum number of information messages.

Default: 50

Description: With this option the amount of information messages can be set.

Example: LINKOPTIONS=-WmsgNi10

Only ten information messages are logged.

See also: Option -WmsgNe

Option -WmsgNw

-WmsgNu

-WmsgNu: Disable User Messages

Group: MESSAGE

Syntax: "-WmsgNu" ["=" {"a" | "b" | "c" | "d"}].

Arguments: “a”: Disable messages about include files

“b”: Disable messages about reading files

“c”: Disable messages about generated files

“d”: Disable messages about processing statistics

“e”: Disable informal messages

Default: none.

Description: The application produces some messages which are not in the normal message categories (WARNING, INFORMATION, WRROR, FATAL). With this option such messages can be disabled. The idea of this option is to reduce the amount of messages and to simplify the error parsing of other tools.

“a”: The application informs about all included files. With this suboption this can be disabled.

“b”: With this suboption messages about reading files e.g. the files used as input can be disabled.

“c”: Disables messages informing about generated files.

“d”: At the end the application may inform about statistics, e.g. code size, RAM/ROM usage and so on. With this suboption this can be disabled.

“e”: With this option informal messages (e.g. memory model, floating point format, ...) can be disabled.

Note: Depending on the application, not all suboptions may make sense. In this case they are just ignored for compatibility.

Example: *-WmsgNu=c*

See also: none.

-WmsgNw

-WmsgNw: Number of Warning Messages

Group: MESSAGE

Syntax: "-WmsgNw" <number>.

Arguments: <number>: Maximum number of warning messages.

Default: 50

Description: With this option the amount of warning messages can be set.

Example: LINKOPTIONS=-WmsgNw15

Only 15 warning messages are logged.

See also: Option -WmsgNe

Option -WmsgNi

-WmsgSd

-WmsgSd: Setting a Message to Disable

Group: MESSAGE

Syntax: "-WmsgSd" <number>.

Arguments: <number>: Message number to be disabled, for example, 1201

Default: none.

Description: With this option a message can be disabled, so it does not appear in the error output.

Example: LINKOPTIONS=-WmsgSd1201

disables the message for no stack declaration.

See also: Option -WmsgSi

Option -WmsgSw

Option -WmsgSe

-WmsgSe

-WmsgSe: Setting a Message to Error

Group: MESSAGE

Syntax: "-WmsgSe" <number>.

Arguments: <number>: Message number to be an error, for example, 1201

Default: none.

Description: Allows changing a message to an error message.

Example: LINKOTIONS=-WmsgSe1201

See also: Option -WmsgSd

Option -WmsgSi

Option -WmsgSw

-WmsgSi

-WmsgSi: Setting a Message to Information

Group: MESSAGE

Syntax: "-WmsgSi" <number>.

Arguments: <number>: Message number to be an information, e.g. 1201

Default: none.

Description: With this option a message can be set to an information message

Example: LINKOPTIONS=-WmsgSi1201

See also: Option -WmsgSd

Option -WmsgSw

Option -WmsgSe

-WmsgSw

-WmsgSw: Setting a Message to Warning

Group: MESSAGE

Syntax: "-WmsgSw" <number>.

Arguments: <number>: Error number to be a warning, for example, 1201

Default: none.

Description: With this option a message can be set to a warning message.

Example: LINKOPTIONS=-WmsgSw1201

See also: Option -WmsgSd

Option -WmsgSi

Option -WmsgSe

-WOutFile

-WOutFile: Create Error Listing File

Group: MESSAGE

Syntax: "-WOutFile" ("On" | "Off").

Arguments: none.

Default: Error listing file is created.

Description: This option controls if a error listing file should be created at all. The error listing file contains a list of all messages and errors which are created during a compilation. Since the text error feedback can now also be handled with pipes to the calling application, it is possible to obtain this feedback without an explicit file. The name of the listing file is controlled by the environment variable ERRORFILE.

Example: -WOutFileOn

The error file is created as specified with ERRORFILE.

-WOutFileOff

No error file is created.

See also: Option -WErrFile

Option -WStdout

-WStdout

-WStdout: Write to standard output

Group: MESSAGE

Syntax: "-WStdout" ("On" | "Off").

Arguments: none.

Default: Output is written to stdout.

Description: With Windows applications, the usual standard streams are available. But text written into them does not appear anywhere unless explicitly requested by the calling application. With this option is can be controlled if the text to error file should also be written into the stdout.

Example: -WStdoutOn

All messages are written to stdout.

-WErrFileOff

Nothing is written to stdout.

See also: Option -WErrFile

Option -WOutFile

6

Linking Issues

Object Allocation

The whole object allocation is performed trough the SEGMENTS(ELF) (or SECTIONS(HIWARE)) and PLACEMENT blocks.

The SEGMENTS Block (ELF)

The SEGMENTS Block is optional, it only increases the readability of the linker input file. It allows to assign meaningful names to contiguous memory areas on the target board. Memory within such an area share common attribute:

- qualifier
- alignment rules
- filling character

Two types of segments can be defined:

- physical segments
- virtual segments

Physical Segments

Physical segments are closely related to hardware memory areas.

For example, there may be one READ_ONLY segment for each bank of the target board ROM area and another one covering the whole target board RAM area.

Example:

For simple memory model you can define a segment for the RAM area and another one for the ROM area.

```
LINK   test.abs
NAMES  test.o startup.o END
SEGMENTS
  RAM_AREA = READ_WRITE 0x00000 TO 0x07FFF;
  ROM_AREA = READ_ONLY  0x08000 TO 0x0FFFF;
END
PLACEMENT
    DEFAULT_RAM        INTO RAM_AREA;
    DEFAULT_ROM        INTO ROM_AREA;
END
STACKSIZE 0x50
```

For banked memory model you can define a segment for the RAM area, another for the non-banked ROM area and one for each target processor bank.

```
LINK   test.abs
NAMES  test.o startup.o END
SEGMENTS
  RAM_AREA        = READ_WRITE 0x00000 TO 0x07FFF;
  NON_BANKED_AREA = READ_ONLY  0x0C000 TO 0x0FFFF;
  BANK0_AREA      = READ_ONLY  0x08000 TO 0x0BFFF;
  BANK1_AREA      = READ_ONLY  0x18000 TO 0x1BFFF;
  BANK2_AREA      = READ_ONLY  0x28000 TO 0x2BFFF;
END
PLACEMENT
    DEFAULT_RAM       INTO RAM_AREA;
    _PRESTART, STARTUP,
    ROM_VAR,
    NON_BANKED, COPY  INTO NON_BANKED_AREA;
    DEFAULT_ROM       INTO BANK0_AREA, BANK1_AREA,
                           BANK2_AREA;
END
STACKSIZE 0x50
```

Virtual Segment

A physical segment may be split into several virtual segments, allowing a better structuring of object allocation and also allowing to take advantage of some processor specific property.

Example:

For HC12 is small memory model you can define a segment for the direct page area, another for the rest of the RAM area and another one for the ROM area.

```
LINK    test.abs
NAMES   test.o startup.o END

SEGMENTS
  DIRECT_RAM = READ_WRITE 0x00000 TO 0x000FF;
  RAM_AREA   = READ_WRITE 0x00100 TO 0x07FFF;
  ROM_AREA   = READ_ONLY  0x08000 TO 0x0FFFF;
END
PLACEMENT
    myRegister         INTO DIRECT_RAM;
    DEFAULT_RAM        INTO RAM_AREA;
    DEFAULT_ROM        INTO ROM_AREA;
END
STACKSIZE 0x50
```

Segment Qualifier

Different qualifiers are available for segments. Table 6.1 describes the available qualifiers:

Table 6.1 Qualifiers and their description

Qualifier	Description
READ_ONLY	Qualifies a segment, where read only access is allowed. Objects within such a segment are initialized at application loading time.
READ_WRITE	Qualifies a segment, where read and write accesses are allowed. Objects within such a segment are initialized at application startup.

Qualifier	Description
NO_INIT	Qualifies a segment, where read and write accesses are allowed. Objects within such a segment remain unchanged during application startup. This qualifier may be used for segments referring to a battery backed RAM. Sections placed in a NO_INIT segment should not contain any initialized variable (variable defined as ‘int c = 8’).
PAGED	Qualifies a segment, where read and write accesses are allowed. Objects within such a segment remain unchanged during application startup. Additionally, objects located in two PAGED segments may overlap. This qualifier is used for memory areas, where some user defined page-switching mechanism is required. Sections placed in a NO_INIT segment should not contain any initialized variable (variable defined as ‘int c = 8’).

NOTE For debugging purposes one sometimes wants to load code into RAM areas. Because this code should be loaded at load time, such areas should be qualified as READ_ONLY.
READ_ONLY means for the linker that such objects are initialized at program load time. The linker does not know (and does not care) if at runtime the target code does write to a READ_ONLY area.

NOTE Anything located in a READ_WRITE segment is initialized at application startup time. So especially the application code, which does this initialization and the data used for this initialization (init, zero out, copy down) cannot be located in a READ_WRITE section, but only in a READ_ONLY section.
The program loader can however at program loading time write the content of READ_ONLY sections into a RAM area.

NOTE If a application does not use any startup code to initialize READ_WRITE sections, then no such sections should be present in the prm file. Instead use NO_INIT sections.

Segment Alignment

The default alignment rule depends on the processor and memory model used. The HC12 processor do not require any alignment for code or data objects. One can choose to define his own alignment rule for a segment. The alignment rule defined for a segment block overrides the default alignment rules associated with the processor and memory model.

The alignment rule has the following format:

```
[defaultAlignment] {"["ObjSizeRange":"alignment"]"}
```

where:

defaultAlignment	is the alignment value for all objects which do not match the conditions of any range defined afterward.
ObjSizeRange	defines a certain condition. The condition is from the form: size: the rule applies to objects, which size is equal to 'size'. < size: the rule applies to objects, which size is smaller than 'size'. > size: the rule applies to objects, which size is bigger than 'size' <= size: the rule applies to objects, which size is smaller or equal to 'size' >= size: the rule applies to objects, which size is bigger or equal to 'size' from size1 to size2: the rule applies to objects, which size is bigger or equal to 'size1'and smaller or equal to 'size2'.
alignment	defines the alignment value for objects matching the condition defined in the current alignment block (enclosed in square bracket).

Example:

```
LINK   test.abs
NAMES  test.o startup.o END

SEGMENTS
  DIRECT_RAM = READ_WRITE 0x00000 TO 0x000FF
               ALIGN 2 [< 2: 1];
  RAM_AREA   = READ_WRITE 0x00100 TO 0x07FFF
               ALIGN [1:1] [2..3:2] [>=4:4];
  ROM_AREA   = READ_ONLY  0x08000 TO 0x0FFFF;
END
PLACEMENT
```

```
    myRegister         INTO DIRECT_RAM;
    DEFAULT_RAM        INTO RAM_AREA;
    DEFAULT_ROM        INTO ROM_AREA;
END
STACKSIZE 0x50
```

In previous example:

- In segment DIRECT_RAM, objects which size is 1 byte are aligned on byte boundary, all other objects are aligned on 2-bytes boundary.
- In segment RAM_AREA, objects which size is 1 byte are aligned on byte boundary, objects which size is equal to 2 or 3 bytes are aligned on 2-bytes boundary, all other objects are aligned on 4-bytes boundary.
- Default alignment rule applies in the segment ROM_AREA.

Segment Fill Pattern

The default fill pattern for code and data segment is the null character. One can choose to define his own fill pattern for a segment The fill pattern definition in the segment block overrides the default fill pattern. Note that the fill pattern is used too to fill up a segment to the segment end boundary.

Example:

```
LINK   test.abs
NAMES  test.o startup.o END

SEGMENTS
  DIRECT_RAM = READ_WRITE 0x00000 TO 0x000FF
               FILL 0xAA;
  RAM_AREA   = READ_WRITE 0x00100 TO 0x07FFF
               FILL 0x22;
  ROM_AREA   = READ_ONLY  0x08000 TO 0x0FFFF;
END
PLACEMENT
    myRegister         INTO DIRECT_RAM;
    DEFAULT_RAM        INTO RAM_AREA;
    DEFAULT_ROM        INTO ROM_AREA;
END
STACKSIZE 0x50
```

In previous example:

- In segment DIRECT_RAM, alignment bytes between objects are initialized with 0xAA.
- In segment RAM_AREA, alignment bytes between objects are initialized with 0x22.
- In segment ROM_AREA, alignment bytes between objects are initialized with 0x00.

The SECTIONS Block (HIWARE + ELF)

The segments block is optional, it only increases the readability of the linker input file. It allows to assign meaningful names to contiguous memory areas on the target board. Memory within such an area share common attribute:

- qualifier,

Two types of segments can be defined:

- physical segments
- virtual segments.

Physical Segments

Physical segments are closely related to hardware memory areas.

For example, there may be one READ_ONLY segment for each bank of the target board ROM area and another one covering the whole target board RAM area.

Example:

For simple memory model you can define a segment for the RAM area and another one for the ROM area.

```
LINK   test.abs
NAMES  test.o startup.o END
SECTIONS
  RAM_AREA = READ_WRITE 0x00000 TO 0x07FFF;
  ROM_AREA = READ_ONLY  0x08000 TO 0x0FFFF;
PLACEMENT
    DEFAULT_RAM         INTO RAM_AREA;
    DEFAULT_ROM         INTO ROM_AREA;
END
STACKSIZE 0x50
```

For banked memory model you can define a segment for the RAM area, another for the non-banked ROM area and one for each target processor bank.

```
LINK   test.abs
NAMES  test.o startup.o END
SECTIONS
  RAM_AREA        = READ_WRITE 0x00000 TO 0x07FFF;
  NON_BANKED_AREA = READ_ONLY  0x0C000 TO 0x0FFFF;
  BANK0_AREA      = READ_ONLY  0x08000 TO 0x0BFFF;
  BANK1_AREA      = READ_ONLY  0x18000 TO 0x1BFFF;
  BANK2_AREA      = READ_ONLY  0x28000 TO 0x2BFFF;
PLACEMENT
    DEFAULT_RAM       INTO RAM_AREA;
    _PRESTART, STARTUP,
    ROM_VAR,
    NON_BANKED, COPY  INTO NON_BANKED_AREA;
    DEFAULT_ROM       INTO BANK0_AREA, BANK1_AREA,
                           BANK2_AREA;
END
STACKSIZE 0x50
```

Virtual Segment

A physical segment may be split into several virtual segments, allowing a better structuring of object allocation and also allowing to take advantage of some processor specific property.

Example:

For HC12 is small memory model you can define a segment for the direct page area, another for the rest of the RAM area and another one for the ROM area.

```
LINK   test.abs
NAMES  test.o startup.o END

SECTIONS
  DIRECT_RAM = READ_WRITE 0x00000 TO 0x000FF;
  RAM_AREA   = READ_WRITE 0x00100 TO 0x07FFF;
  ROM_AREA   = READ_ONLY  0x08000 TO 0x0FFFF;
PLACEMENT
    myRegister        INTO DIRECT_RAM;
    DEFAULT_RAM       INTO RAM_AREA;
    DEFAULT_ROM       INTO ROM_AREA;
```

```
END
STACKSIZE 0x50
```

Segment Qualifier

Different qualifiers are available for segments. Table 6.2 describes the available qualifiers:

Table 6.2 Qualifiers and their description

Qualifier	Meaning
READ_ONLY	Qualifies a segment, where read only access is allowed. Objects within such a segment are initialized at application loading time.
CODE (ELF)	Qualifies a code segment in a harvard architecture in the ELF object file format. For cores with Von Neumann Architecture (combined code and data address space) or for the HIWARE object file format use READ_ONLY instead.
READ_WRITE	Qualifies a segment, where read and write accesses are allowed. Objects within such a segment are initialized at application startup.
NO_INIT	Qualifies a segment, where read and write accesses are allowed. Objects within such a segment remain unchanged during application startup. This qualifier may be used for segments referring to a battery backed RAM. Sections placed in a NO_INIT segment should not contain any initialized variable (variable defined as ‘int c = 8’).
PAGED	Qualifies a segment, where read and write accesses are allowed. Objects within such a segment remain unchanged during application startup. Additionally, objects located in two PAGED segments may overlap. This qualifier is used for memory areas, where some user defined page-switching mechanism is required. Sections placed in a NO_INIT segment should not contain any initialized variable (variable defined as ‘int c = 8’).

NOTE	For debugging purposes one sometimes wants to load code into RAM areas. Because this code should be loaded at load time, such areas should be qualified as READ_ONLY. READ_ONLY means for the linker that such objects are initialized at program load time. The linker does not know (and does not care) if at runtime the target code does write to a READ_ONLY area.

NOTE	Anything located in a READ_WRITE segment is initialized at application startup time. So especially the application code, which does this initialization and the data used for this initialization (init, zero out, copy down) cannot be located in a READ_WRITE section, but only in a READ_ONLY section. The program loader can however at program loading time write the content of READ_ONLY sections into a RAM area.

NOTE	If a application does not use any startup code to initialize READ_WRITE sections, then no such sections should be present in the prm file. Instead use NO_INIT sections.

PLACEMENT Block

The placement block allows to physically place each section from the application in a specific memory area (segment). The sections specified in a PLACEMENT block may be linker-predefined sections or user sections specified in one of the source file building the application.

A programmer may decide to organize his data into sections:

- to increase structuring of the application
- to ensure that common purpose data are grouped together
- to take advantage of target processor specific addressing mode.

Specifying a List of Sections

When several sections are specified on a PLACEMENT statement, the sections are allocated in the sequence they are enumerated.

Example:

```
LINK   test.abs
NAMES  test.o startup.o END

SECTIONS
  RAM_AREA   = READ_WRITE 0x00100 TO 0x002FF;
  STK_AREA   = READ_WRITE 0x00300 TO 0x003FF;
  ROM_AREA   = READ_ONLY  0x08000 TO 0x0FFFF;
PLACEMENT
    DEFAULT_RAM, dataSec1,
     dataSec2              INTO RAM_AREA;
    DEAFULT_ROM, myCode INTO ROM_AREA;
    SSTACK                 INTO STK_AREA;
END
```

In previous example:

- Inside of segment RAM_AREA, the objects defined in the section .data are allocated first, then the objects defined in section dataSec1 then objects defined in section dataSec2.
- Inside of segment ROM_AREA, the objects defined in section .text are allocated first, then the objects defined in section myCode

NOTE As the linker is case sensitive, the name of the sections specified in the PLACEMENT block must be valid predefined or user defined section. For the linker the sections DataSec1 and dataSec1 are two different sections

Specifying a List of Segments

When several segments are specified on a PLACEMENT statement, the segments are used in the sequence they are enumerated. Allocation is performed in the first segment in the list, until this segment is full. Then allocation continues on the next segment in the list, an so on until all objects are allocated.

Example:

```
LINK   test.abs
 NAMES  test.o startup.o END
 SECTIONS
   RAM_AREA         = READ_WRITE 0x00100 TO 0x002FF;
   STK_AREA         = READ_WRITE 0x00300 TO 0x003FF;
   NON_BANKED_AREA = READ_ONLY  0x0C000 TO 0x0FFFF;
```

```
  BANK0_AREA      = READ_ONLY  0x08000 TO 0x0BFFF;
  BANK1_AREA      = READ_ONLY  0x18000 TO 0x1BFFF;
  BANK2_AREA      = READ_ONLY  0x28000 TO 0x2BFFF;
PLACEMENT
    DEFAULT_RAM         INTO RAM_AREA;
    SSTACK              INTO STK_AREA;
    _PRESTART, STARTUP,
    ROM_VAR,
    NON_BANKED, COPY    INTO NON_BANKED_AREA;
    DEFAULT_ROM         INTO BANK0_AREA, BANK1_AREA,
                             BANK2_AREA;
END
```

In previous example:

- Functions implemented in section .text are allocated first in segment BANK0_AREA. When there is not enough memory available in this segment, allocation continues in segment BANK_1_AREA, then in BANK2_AREA

NOTE As the linker is case sensitive, the name of the segments specified in the PLACEMENT block must be valid segment names defined in the SEGMENTS block. For the linker the segments Ram_Area and RAM_AREA are two different segments.

Allocating User Defined Sections (ELF)

All sections do not need to be enumerated in the placement block. The segments where sections, which do not appear in the PLACEMENT block, are allocated depends on the type of the section.

- Sections containing data are allocated next to the section.data.
- Sections containing code, constant variables or string constants are allocated next to the section .text.

Allocation in the segment where .data is placed is performed as follows:

- Objects from section .data are allocated
- Objects from section .bss are allocated (if .bss is not specified in the PLACEMENT block).
- Objects from the first user defined data section, which is not specified in the PLACEMENT block, are allocated.

- Objects from the next user defined data section, which is not specified in the PLACEMENT block, are allocated.
- and so on until all user defined data sections are allocated.
- If the section .stack is not specified in the PLACEMENT block and is defined with a STACKSIZE command, the stack is allocated then.

.data	.bss	User Data 1	. . .	User Data n	stack

Allocation in the segment where .text is placed is performed as follows:

- Objects from section .init are allocated (if .init is not specified in the PLACEMENT block).
- Objects from section .startData are allocated (if .startData is not specified in the PLACEMENT block).
- Objects from section .text are allocated
- Objects from section .rodata are allocated (if .rodata is not specified in the PLACEMENT block).
- Objects from section .rodata1 are allocated (if .rodata1 is not specified in the PLACEMENT block).
- Objects from the first user defined code section, which is not specified in the PLACEMENT block, are allocated.
- Objects from the next user defined code section, which is not specified in the PLACEMENT block, are allocated.
- and so on until all user defined code sections are allocated.
- Objects from section .copy are allocated (if .copy is not specified in the PLACEMENT block).

.init	.startData	.text	.rodata	.rodata1	User Code1	...	User Code n	.copy

Allocating User Defined Sections (HIWARE)

All sections do not need to be enumerated in the placement block. The segments where sections, which do not appear in the PLACEMENT block, are allocated depends on the type and attributes of the section.

- Sections containing code are allocated next to the section DEFAULT_ROM.
- Sections containing constants only are allocated next to the section DEFAULT_ROM. This behavior can be changed with option -CRam.
- Sections containing string constants are allocated next to the section DEFAULT_ROM.
- Sections containing data are allocated next to the section DEFAULT_RAM.

Allocation in the segment where DEFAULT_RAM is placed is performed as follows:

- Objects from section DEFAULT_RAM are allocated
- If the option -CRam is specified, Objects from section ROM_VAR are allocated, if ROM_VAR is not mentioned in the PLACEMENT block.
- Objects from user defined data sections, which are not specified in the PLACEMENT block, are allocated. If option -CRam is specified, constant sections are allocated together with non constant data sections.
- If the section SSTACK is not specified in the PLACEMENT block and is defined with a STACKSIZE command, the stack is allocated then.

DEFAULT_RAM	User Data 1	...	User Data n	SSTACK

Allocation in the segment where DEFAULT_ROM is placed is performed as follows:

- Objects from section _PRESTART are allocated (if _PRESTART is not specified in the PLACEMENT block).
- Objects from section STARTUP are allocated (if STARTUP is not specified in the PLACEMENT block).
- Objects from section ROM_VAR are allocated (if ROM_VAR is not specified in the PLACEMENT block). If option -CRam is specified, ROM_VAR is allocated in the RAM.
- Objects from section SSTRING (string constants) are allocated (if SSTRING is not specified in the PLACEMENT block).
- Objects from section DEFAULT_ROM are allocated
- Objects from all user defined code sections and constant data sections, which are not specified in the PLACEMENT block, are allocated.
- Objects from section COPY are allocated (if .copy is not specified in the PLACEMENT block).

_PRESTART	STARTUP	ROM_VAR	SSTRING	DEFAULT_ROM	User Code 1	...	User Code n	COPY

Initializing Vector Table

Vector table initialization is performed using the VECTOR command.

VECTOR Command

This command is specially defined to initialize the vector table.

The syntax “VECTOR <Number>” can be used. In this case the Linker allocates the vector depending on the target CPU. The vector number zero is usually the reset vector, but depends on the target. The Linker knows about the default start location of the vector table for each target supported.

The Syntax VECTOR ADDRESS can be used as well. The size of the entries in the vector table depends on the target processor.

Different syntax are available for the VECTOR command. Table 6.3 describes the VECTOR command syntax.

Table 6.3 VECTOR command syntax and their description

Command	Description
VECTOR ADDRESS 0xFFFE 0x1000	indicates that the value 0x1000 must be stored at address 0xFFFE
VECTOR ADDRESS 0xFFFE FName	indicates that the address of the function FName must be stored at address 0xFFFE.
VECTOR ADDRESS 0xFFFE FName OFFSET 2	indicates that the address of the function FName incremented by 2 must be stored at address 0xFFFE

The last syntax may be very useful, when working with a common interrupt service routine.

Smart Linking (ELF)

Because of smart linking, only the objects referenced are linked with the application. The application entry points are:

- The application init function
- The main function
- The function specified in a VECTOR command.

All the previously enumerated entry points and the objects they referenced are automatically linked with the application.

The customer can specify additional entry points using the command ENTRIES in the prm file.

Mandatory Linking from an Object

One can choose to link some non-referenced objects in his application. This may be useful to ensure that a software version number is linked with the application and stored in the final product EPROM.

This may also be useful to ensure that a vector table, which has been defined as a constant table of function pointers is linked with the application.

Example:

```
ENTRIES
  myVar1 myVar2 myProc1 myProc2
END
```

In previous example:

- The variables myVar1 and myVar2 as well as the function myProc1 and myProc2 are specified to be additional entry points in the application

NOTE As the linker is case sensitive, the name of the objects specified in the ENTRIES block must be objects defined somewhere in the application. For the linker the variable MyVar1 and myVar1 are two different objects.

Mandatory Linking from all Objects defined in a File

One can choose to link all objects defined in a specified object file in his application.

Example:

```
ENTRIES
  myFile1.o:* myFile2.o:*
END
```

In previous example:

- All the objects (functions, variables, constant variables or string constants) defined in file myFile1.o and myFile2.o are specified to be additional entry points in the application.

Switching OFF Smart Linking for the Application

One can choose to switch OFF smart linking. All objects are linked in the application.

Example:

```
ENTRIES
  *
END
```

In previous example:

- Smart linking is switched OFF for the whole application. That means that all objects defined in one of the binary file building the application are linked with the application.

Smart Linking (HIWARE + ELF)

Because of smart linking, only the objects referenced are linked with the application. The application entry points are:

- The application init function

- The main function
- The function specified in a VECTOR command.

All the previously enumerated entry points and the objects they referenced are automatically linked with the application.

The customer can specify additional entry points using the command ENTRIES in the prm file.

Mandatory Linking from an Object

One can choose to link some non-referenced objects in his application. This may be useful to ensure that a software version number is linked with the application and stored in the final product EPROM.

This may also be useful to ensure that a vector table, which has been defined as a constant table of function pointers is linked with the application.

Example:

```
ENTRIES
  myVar1 myVar2 myProc1 myProc2
END
```

In previous example:

- The variables myVar1 and myVar2 as well as the function myProc1 and myProc2 are specified to be additional entry points in the application

NOTE As the linker is case sensitive, the name of the objects specified in the ENTRIES block must be objects defined somewhere in the application. For the linker the variable MyVar1 and myVar1 are two different objects.

Mandatory Linking from all Objects defined in a File

One can choose to link all objects defined in a specified object file in his application. In that purpose, you only need to specify a ‘+’ after the name of the module in the NAMES block.

Example:

```
NAMES
  myFile1.o+ myFile2.o+ start.o ansi.lib
END
```

In previous example:

- All the objects (functions, variables, constant variables or string constants) defined in file myFile1.o and myFile2.o are specified to be additional entry points in the application.

Binary Files building an Application (ELF)

The names of the binary files building an application may be specified in the NAMES block or in the ENTRIES block. Usually a NAMES block is sufficient.

NAMES Block

The list of all the binary files building the application are usually listed in the NAMES block. Additional binary files may be specified by the option -add. If all binary files should be specified by the command line option -add, then an empty NAMES block (just NAMES END) must be specified.

Example:

```
NAMES
  myFile1.o myFile2.o
END
```

In previous example:

- The binary files myFile1.o and myFile2.o build the application.

ENTRIES Block

If a file name is specified in the ENTRIES block, the corresponding file is considered to be part of the application, even if it does not appear in the NAMES block. The file specified in the ENTRIES block may also be present in the NAMES block. Name from absolute, ROM library or library files are not allowed in the ENTRIES block.

Example:

```
LINK    test.abs
NAMES   test.o startup.o END

SEGMENTS
  DIRECT_RAM = READ_WRITE 0x00000 TO 0x000FF;
  STK_AREA   = READ_WRITE 0x00200 TO 0x002FF;
  RAM_AREA   = READ_WRITE 0x00300 TO 0x07FFF;
  ROM_AREA   = READ_ONLY  0x08000 TO 0x0FFFF;
END
PLACEMENT
    myRegister        INTO DIRECT_RAM;
    DEFAULT_RAM       INTO RAM_AREA;
    DEFAULT_ROM       INTO ROM_AREA;
    SSTACK            INTO STK_AREA;
END
ENTRIES
  test1.o:* test.o:*
END
```

In previous example:

- The file test.o, test1.o and startup.o build the application. All objects defined in the module test1.o and test.o will be linked with the application.

Binary Files building an Application (HIWARE)

The names of the binary files building an application may be specified in the NAMES block or in the ENTRIES block. Usually a NAMES block is sufficient.

NAMES Block

The list of all the binary files building the application are usually listed in the NAMES block. Additional binary files may be specified by the option -add. If all binary files should be specified by the command line option -add, then an empty NAMES block (just NAMES END) must be specified.

Example:

```
NAMES
  myFile1.o myFile2.o
END
```

In previous example:

- The binary files myFile1.o and myFile2.o build the application.

Allocating Variables in "OVERLAYS"

When your application consist in two distinct parts (or execution unit), which are never activated at the same time, you can ask the linker to overlap the global variables of both parts. For this purpose you should pay attention to the following points in your application source files:

- The global variable from the different parts must be defined in separate data segments. Do not use the same segment for both execution units.
- The global variables in both execution units must not be defined with initializer, but should be initialized using assignments in the application source code.

In the prm file, you can then define two distinct memory areas with attribute PAGED. Memory areas with attributes PAGED are not initialized during startup. For this reason they cannot contain any variable defined with initializer. The linker will not perform any overlap check on PAGED memory areas.

Example:

In your source code support you have two execution unit: APPL_1 and APPL_2.

- All global variables from APPL_1 are defined in segment APPL1_DATA_SEG
- All global variables from APPL_2 are defined in segment DEFAULT_RAM and APPL2_DATA_SEG

The prm file will look as follows:

```
LINK test.abs

NAMES  test.o appl1.o appl2.o startup.o END

SECTIONS
    MY_ROM   = READ_ONLY  0x800 TO 0x9FF;
```

```
    MY_RAM_1 = PAGED      0xA00 TO 0xAff;
    MY_RAM_2 = PAGED      0xA00 TO 0xAff;
    MY_STK   = READ_WRITE 0xB00 TO 0xBFF;

PLACEMENT
    DEFAULT_ROM      INTO  MY_ROM;
    DEFAULT_RAM,
    APPL2_DATA_SEG   INTO  MY_RAM_2;
    APPL1_DATA_SEG   INTO  MY_RAM_1;
    SSTACK           INTO  MY_STK;  /* Stack cannot be allocated in a
PAGED
memory area. */
END
```

Overlapping Locals

This section is only for targets which do allocated local variables like global variables at fixed addresses.

Some small targets do not have a stack for local variables. So the compiler uses pseudo-statically objects for local variables. In contrast to other targets which allocate such variables on the stack, these variables must then be allocated by the linker. On the stack multiple local variables are automatically allocated at the same address at a different time. A similar overlapping scheme is implemented by the linker to save memory for local variables.

Example:

```
void f(void) { long fa; ....; }
void g(void) { long ga; ....; }
void main(void) { long lm; f(); g(); }
```

In the example above, the functions f and g are never active at the same time. Therefore the local variable fa and ga can be allocated at the same address.

NOTE When local variables are allocated at fixed addresses, the resulting code is not reentrant. One function must be called only once at a time. Special care has to be taken about interrupt functions. They must not call any function which might be active at the interrupt time.

To be on the save side, usually interrupt functions are using a different set of functions than non interrupt functions.

NOTE For the view of the linker, parameter and spill objects do not differ from local variables. All these objects are allocated together.

The linker analyses the call graph of one root function at a time and allocates all local variables used by all depending functions at this time. Variables depending on different root functions are allocated non-overlapping except in the special case of an OVERLAP_GROUP(ELF).

Algorithm

Algorithm for the overlap allocation is quite simple:

1. If current object depends on other objects first allocate the dependents
2. Calculate the maximum address used by any dependent object. If none exist, use the base reserved for the current root.
3. Allocate all locals starting at the maximum.

This algorithm is called for all roots. The base of the root is first calculated as the maximum used so far.

Example

```
void g(long g_par)  { }
void h(long l_par)  { }
void main(void)  {
  char ch;
  g(1);
  h(2);
}
void interrupt 1 inter(void) {
  long inter_loc;
}
```

The function main is a root because it is the application main function and inter is a root because it is called by a interrupt.

```
...
```

```
SECTIONS
...
  OVERLAP_RAM = NO_INIT 0x0060 TO 0x0068;
...
PLACEMENT
...
    _OVERLAP          INTO OVERLAP_RAM;
...
END
```

NOTE In the ELF object file format the name "_OVERLAP" is a synonym for the ".overlap" segment.

<table>
<tr><td>0x60</td><td>0x61</td><td>0x62</td><td>0x63</td><td>0x64</td><td>0x65</td><td>0x66</td><td>0x67</td><td>0x68</td></tr>
<tr><td colspan="4">g_par</td><td>ch</td><td colspan="4">inter_loc</td></tr>
<tr><td colspan="4">l_par</td><td colspan="5"></td></tr>
</table>

The algorithm is started with main. As h and g depend on main, their parameters g_par and l_par are allocated starting at address 0x60 in the _OVERLAP segment. Next the local ch is allocated at 0x64 because all lower addresses were already used by dependents. After main was finished, the base for the second root is calculated as 0x65, where inter_loc is also allocated.

The following items are considered as root points for the overlapping allocation in the ELF object file format:

- objects specified in a DEPENDENCY ROOT block
- objects specified in a OVERLAP_GROUP block
- application main function (specified with prm file entry MAIN) and application entry point (specified with prm file entry INIT)
- objects specified in a ENTRIES block
- absolute objects
- interrupt vectors
- All objects in non Smart Linked object files.

NOTE The main function (main) and the application entry point (_Startup) are implicitly defined as one OVERLAP_GROUP. In the startup code delivered with the compiler, this saves about 8 bytes because the locals of Init, Copy and main are overlapped. When _Startup itself is changed and now also needs locals which must be alive over the call to main, define the _Startup function as single entry in an OVERLAP_GROUP:
OVERLAP_GROUP _Startup END

The overlap section _OVERLAP (in ELF also named .overlap) must be allocated in a NO_INIT area. The section _OVERLAP cannot be split into several areas.

Name Mangling for Overlapping Locals

When parameters are passed on the stack, then the matching of the callers and the callee arguments works by their position on the stack. For overlapped locals (which do include parameters not passed in registers as well), the matching is done by the linker using the parameter name.

Consider the following example:

```
void callee(long i);
void caller(void) {
  callee(1);
}
void callee(long k) {
}
```

The name i of the declaration of callee does not match the name used in the definition. Actually, the declaration might not specify a name at all. As the link between the caller and callees argument is done by the name, they both have to use the same name. Because of this, the compiler does generate an artificial name for the callee's parameter _calleep0. This name is built starting with an underscore ("_"), then appending the function name, a "p" and finally the number of the argument.

NOTE In ELF, there is a second name mangling needed to encode the name of the defining function into its name. For details, see below.

Compiler users do not need to know about the name mangling at all. The compiler does it for them automatically.

However, if you want to write functions with overlapping locals in assembler, then you have to do the name mangling yourself. This is especially important if you are calling C functions from assembler code or assembler functions from C code.

Name Mangling in the ELF Object File Format

In the ELF Object File Format, there is no predefined way to specify to which function an actual parameter does belong. So the compiler does some special name mangling which adds the name of the function into the link time name.

In ELF, the name is build the following way:

If the object is a function parameter, use a "p" followed by the number of the argument instead of the object name given in the source file.

1. prefix "__OVL_"
2. If the function name contains a underscore ("_"), the number of characters of the function name followed by a underscore ("_"). Nothing if the function name does not contain an underscore.
3. The function name.
4. An underscore ("_").
5. If the object name contains a underscore ("_"), the number of characters of the object followed by one underscore ("_"). Nothing if the object name does not contain an underscore.
6. The object name.

Example (ELF):

```
void f(long p) {
  char a;
  char b_c;
}
```

Does generate the following mangled names

```
  p:    "__OVL_f_p0"    (HIWARE format: "_fp0")
  a:    "__OVL_f_a"     (HIWARE format: "a")
b_c:    "__OVL_f_3_b_c" (HIWARE format: "b_c")
```

Defining an function with overlapping parameters in Assembler

This section covers advanced topics which are only important if you plan to write assembler functions using a C calling convention with overlapping parameters.

As example, we want to define the function callee:

```
void callee(long k) {
  k= 0;
}
```

In assembler, first the parameter must be defined with its mangled name. The parameter must be in the section _OVERLAP:

```
_OVERLAP: SECTION
callee_p1: DS 4
```

NOTE The _OVERLAP section is often allocated in a short segment. If so, use "_OVERLAP: SECTION SHORT" to specify this.

Next we define the function itself.

```
callee_code: SECTION
callee:
  CLEAR callee_p1,4
  RETURN
```

To avoid processor specific examples, we assume that there is an assembler macro CLEAR which writes as many zero bytes as its second argument to the address specified by its first argument. The second macro RETURN should just generate a return instruction for the actually used processor. The implementations of these two macros are processor specific and not contained in this linker manual.

Finally, we have to export callee and its argument:

```
 XDEF callee
 XDEF callee_p1
```

The whole example in one block:

```
;Processor specific macro definition, please adapt to your target
CLEAR:       MACRO
```

```
            ...
            ENDM

RETURN:     MACRO
            ...
            ENDM

_OVERLAP:   SECTION
callee_p1:  DS 4

callee_code: SECTION

callee:
            CLEAR callee_p1,4
            RETURN
; export function and parameter
            XDEF callee
            XDEF callee_p1
```

Some additional points to consider

- In the ELF format, the name of the p1 parameter must be _OVL_callee_p1 instead of callee_p1.

  Example for ELF:

```
_OVERLAP:       SECTION
_OVL_callee_p1: DS 4

callee_code:    SECTION

callee:
                CLEAR _OVL_callee_p1,4
                RETURN
; export function and parameter
            XDEF callee
            XDEF _OVL_callee_p1
```

- Every function defined in assembler should be in a separate section as a linker section containing code corresponds to a compiler function.

  Example two functions put into one segment:

```
            XDEF callee0
            XDEF callee1
```

```
_OVERLAP:     SECTION
loc0:         DS 4
loc1:         DS 4

code_seg: SECTION
callee0:
              CLEAR loc0,4
              RETURN
callee1:     ; ERROR function should be in separate segment
              CLEAR loc1,4
              RETURN
```

Because callee0 and callee1 are in the same segment, the linker treats them as if they were two entry points of the same function. Because of this, loc0 and loc1 will not be overlapped and additional dependencies are generated.

To solve the problem, put the two functions into separate segments:

```
              XDEF callee0
              XDEF callee1
_OVERLAP:     SECTION
loc0:         DS 4
loc1:         DS 4

code_seg0: SECTION
callee0:
              CLEAR loc0,4
              RETURN
code_seg1: SECTION
callee1:
              CLEAR loc1,4
              RETURN
```

- Parameter objects are exported if the corresponding function is exported too. Locals are usually not exported.

  Example of an illegal non exported definition of a parameter:

```
              XDEF callee
_OVERLAP:     SECTION
callee_p1:    DS 4

callee_code: SECTION

callee:
              CLEAR callee_p1,4
```

```
              RETURN
```

Because callee_p1 is not exported, an external caller of callee will not use the correct actual parameter. Actually, the application will not be able to link because of the unresolved external callee_p1.

To correct it, export callee_p1 too:

```
              XDEF callee
              XDEF callee_p1
_OVERLAP:     SECTION
callee_p1:    DS 4

callee_code: SECTION

callee:
              CLEAR callee_p1,4
              RETURN
```

- Do only use parameters of functions which are actually called. Do not use local variables of other functions. The assembler does not prevent the usage of locals, which would not have been possible in C. Such additional usages are not taken into account for the allocation and may therefore not work as expected. As rule, only access objects defined in the _OVERLAP section from one single SECTION unless the object is a parameter. Parameters can be safely accessed from all sections containing calls to the callee and from the section defining the callee.

  Example of an illegal usage of a local variable

```
_OVERLAP:      SECTION
loc:           DS 4

callee0_code:  SECTION
callee0:
             CLEAR loc,4 ; error: usage of local var loc from two functs
               RETURN

callee1_code:  SECTION
callee1:
             CLEAR loc,4 ; error: usage of local var loc from two functs
               RETURN
```

Instead use two different locals for two different functions:

```
_OVERLAP:      SECTION
```

```
loc0:         DS 4; local var of function callee0
loc1:         DS 4; local var of function callee1

callee0_code: SECTION
callee0:
              CLEAR loc0,4 ; OK, only callee 0 uses loc0
              RETURN

callee1_code: SECTION
callee1:
              CLEAR loc1,4 ; OK, only callee 0 uses loc1
              RETURN
```

- In the HIWARE format, functions defined in assembly *must* access all its parameters and locals allocated in the _OVERLAP segment.

  There must be no unused parameters in the _OVERLAP segment. If this rule is violated, then the linker allocates the parameter in the overlap area of one of the callers. This object can then overlap with the local variables of other callers.

  In the ELF format, the binding to the defining function is done by the name mangling and this restriction does therefore not exists.

  The following example does not work in the HIWARE format because callee_p1 is not accessed.

```
_OVERLAP:    SECTION
callee_p1:   DS 4; error: parameter MUST be accessed

callee_code: SECTION
callee:
  RETURN

To correct it, do use the parameter even if the usage is not actually
necessary:
_OVERLAP:    SECTION
callee_p1:   DS 4; OK parameter is accessed

callee_code: SECTION
callee:
             CLEAR callee_p1,1
             RETURN
```

DEPENDENCY TREE in the Map File

The DEPENDENCY TREE section in the map file was especially built to provide useful information about the overlapped allocation.

Example:

```
volatile int intPending; /* interrupt being handled? */

void interrupt 1 inter(void) {
  int oldIntPending=intPending;
  intPending=TRUE;
  while (0 == read((void*)0x1234)) {}
  intPending=oldIntPending;
}

unsigned char read(void* adr)  {
  return *(volatile char*)adr;
}
```

Does generate the following tree:

```
_Vector_1            : 0x808..0x80B
|
+* inter                : 0x808..0x80B
   | +* oldIntPending        : 0x80A..0x80B
   |
   +* read                 : 0x808..0x809
        +* _readp0              : 0x808..0x809
```

Vector_1 is for the interrupt vector 1 specified in the C source.

The parameter name adr is encoded to _readp0 because in C, parameter names may have different names in different declarations, or even no name as in the example.

Vector_1, inter and read do all depend on the adr parameter of read, which is allocated at 0x808 to 0x809 (inclusive). So this area is included for all these objects. Only Vector_1 and inter do depend on oldIntPending, so the area 0x80A to 0x80B is only contained in these functions.

Optimizing the overlap size

The area of memory used by one function is the area of this function plus the maximum of the areas of all used functions. The branches with the maximum area are marked with a star “*”.

When a local variable is added to a function with a “*”, the whole overlap area will grow by the variable size. More useful, when a variable of a function marked with a “*” is removed, then the size of the overlap may decrease (it may also not, because there can be several functions with a * on the same level). When a marked function is using some variables of its own, then splitting this function into several parts may also reduce the overlap area.

Recursion Checks

Assume, that for the previous example, a second interrupt function exists:

Example

```
void interrupt 2 inter2(void) {
  int oldIntPending=intPending;
  intPending=TRUE;
  while (0 == read((void*)0x1235)) {}
  intPending=oldIntPending;
}
```

Now, there are two dependency trees in the map file

```
 _Vector_2            : 0x808..0x80B
 |
 +* inter2              : 0x808..0x80B
    | +* oldIntPending        : 0x80A..0x80B
    |
    +* read                 : 0x808..0x809
         +* _readp0             : 0x808..0x809

 _Vector_1            : 0x80C..0x80D
 |
 +* inter               : 0x80C..0x80D
    | +* oldIntPending        : 0x80C..0x80D
    |
   +* read                 : 0x808..0x809 (see above) (object allocated
in area of another root)
```

The subtree of the read function is printed only once. The second time, the "(see above)" is printed instead of the whole subtree. The second remark "(object allocated in area of another root)" is more serious. Both interrupt functions are using the same read function. If one interrupt handler can interrupt the other handler, then the parameter of the read functions may be overwritten, the first handler would fail. But if both interrupt are exclusive, which is common for the small processors using overlapped variables, then this information should be added to the prm file to allow an optimal allocation.

Example (prm file):

```
DEPENDENCY
  ROOT inter inter2 END
END
```

Now the warning disappears and both inter and inter2 are contained in the same tree:

```
 DEPENDECY ROOT
 |
 +* inter2                : 0x808..0x80B
 |  | +* oldIntPending        : 0x80A..0x80B
 |  |
 |  +* read                  : 0x808..0x809
 |       +* _readp0              : 0x808..0x809
 |
 +* inter                 : 0x808..0x80B
    | +* oldIntPending        : 0x80A..0x80B
    |
    +* read                  : 0x808..0x809 (see above)
```

Because the oldIntPending's of both handlers are now allocated overlapping, this saves 2 bytes in this example.

NOTE	Vector_1 and Vector_2 are still handled by the linker as additional roots. But because all is allocated using the DEPENDENCY ROOT, the have no influence on the generated code. But their trees are still listed in the DEPENDENCY TREE section in the map file. These trees can be safely ignored.

See Also

ROM Libraries and Overlapping Locals

DEPENDENCY command

OVERLAP_GROUP command

ENTRIES command

Linker Defined Objects

The linker supports to define special objects in order to get the address and size of sections at link time. Objects to be defined by the linker must have a special prefix. Their name must start with one of the strings below and they must not be defined by the application at all.

NOTE	Because the linker defines C variables automatically when their size is known, the usual variables declaration fails for this feature. For an “extern int __SEG_START_SSTACK;”, the linker allocates the size of an int, and does not define the object as address of the stack. Instead use the following syntax so that the compiler/linker has no size for the object: “extern int __SEG_START_SSTACK[];”.

Usual applications of this feature are the initialization of the stack pointer and to get the last address of an application to compute a code checksum at runtime.

The object name is built by using a special prefix and then the name of the symbol.

The following tree prefixes are supported:

- “__SEG_START_”: start address of the segment
- “__SEG_END_”: end address of the segment
- “__SEG_SIZE_”: size of the segment

NOTE	The “__SEG_END_” end address is the address of the first byte behind the named segment.

The remaining text after the prefix is taken as segment name by the linker. If the linker does not find such a segment, a warning is issued an 0 is taken as address of this object.

Because identifiers in C must not contain a period in their name, the HIWARE format aliases can be used for the special ELF names (for example, “SSTACK” instead of “.stack”).

Example:

With the following C source code:

```
#define __SEG_START_REF(a)  __SEG_START_ ## a
#define __SEG_END_REF(a)    __SEG_END_   ## a
#define __SEG_SIZE_REF(a)   __SEG_SIZE_  ## a

#define __SEG_START_DEF(a)  extern char __SEG_START_REF(a) []
#define __SEG_END_DEF(a)    extern char __SEG_END_REF(  a) []
#define __SEG_SIZE_DEF(a)   extern char __SEG_SIZE_REF( a) []

/* To use this feature, first define the symbols to be used: */
  __SEG_START_DEF(SSTACK); // start of stack
  __SEG_END_DEF(SSTACK);   // end of stack
  __SEG_SIZE_DEF(SSTACK);  // size of stack

/* Then use the new symbols with the _REF macros: */
int error;
void main(void) {
  char* stackBottom= (char*)__SEG_START_REF(SSTACK);
  char* stackTop   = (char*)__SEG_END_REF(SSTACK);
  int stackSize= (int)__SEG_SIZE_REF(SSTACK);
  error=0;
  if (stackBottom+stackSize != stackTop) { // top is bottom + size
    error=1;
  }
  for (;;); /* wait here */
}
```

And the following corresponding prm file (must be adapted for some processors):

```
LINK example.abs
  NAMES  example.o  END
SECTIONS
    MY_RAM = READ_WRITE 0x0800 TO 0x0FFF;
    MY_ROM = READ_ONLY  0x8000 TO 0xEFFF;
    MY_STACK = NO_INIT 0x400 TO 0x4ff;
END
PLACEMENT
    DEFAULT_ROM INTO MY_ROM;
    DEFAULT_RAM INTO MY_RAM;
```

```
    SSTACK       INTO MY_STACK;
END
INIT main
```

The linker defined symbols are defined the following way:

```
__SEG_START_SSTACK   0x400
__SEG_END_SSTACK     0x500
__SEG_SIZE_SSTACK    0x100
```

NOTE To use the same source code with other linkers or old linkers, define the symbols in a separate module for them.

NOTE In C, you must use the address as value, and not any value stored in the variable. So in the previous example, "(int)__SEG_SIZE_REF(SSTACK)" was used to get the size of the stack segment and not a C expression like "__SEG_SIZE_REF(SSTACK)[0]".

Automatic Distribution of Paged Functions

One common problem with applications distributed in several pages is how to distribute the functions into the pages. The simple approach is to compile all function calls so that they can take place across page boundaries. Then the linker can distribute the functions without any restrictions.

The disadvantage of this conservative approach is that functions, which are only used within one page would not actually need the paged calling convention. Compiling these functions with a intrapage calling convention does both save memory and execution time. But to guarantee that all calls to an optimized function are within one page, all callers and the callee have to be allocated in a special segment, which is allocated in one single page. The callee's calling convention must be additionally marked as "intrapage".

Example:

C Source:

```
#pragma CODE_SEG FUNCTIONS
void f(void) { ... }
void g(void) { ... f(); ... }
void h(void) { ... g(); ... }
```

Link parameter File:

```
SECTIONS
...
    MY_ROM0  = READ_ONLY 0x06000 TO 0x07FFF;
    MY_ROM1  = READ_ONLY 0x18000 TO 0x18FFF;
    MY_ROM2  = READ_ONLY 0x28000 TO 0x28FFF;
...
PLACEMENT
...
    FUNCTIONS INTO MY_ROM1, MY_ROM2;
...
```

Assume that f and g have place in MY_ROM1. The function h is too large and therefore allocated in MY_ROM2. Further assume for now that f is only called by g.

Even in this simple case, the compiler does not know that f and g are on the same page, so the compiler has to use a page crossing calling convention to call f. Because this is not really needed, the source can be adapted:

```
#define __INTRAPAGE__ .../* actually name depends on the */
             /* target processor. E.g. __near, __far,... */

#pragma CODE_SEG F_AND_G_FUNCTIONS
void __INTRAPAGE__ f(void) { ... }
void g(void) { ... f(); ... }
#pragma CODE_SEG FUNCTIONS
void h(void) { ... g(); ... }
```

Link parameter File:

```
...
    MY_ROM1  = READ_ONLY  0x18000 TO 0x18FFF;
    MY_ROM2  = READ_ONLY  0x28000 TO 0x28FFF;
...
PLACEMENT
..
    F_AND_G_FUNCTIONS INTO MY_ROM1;
    FUNCTIONS INTO MY_ROM2;
```

```
...
```

Now the compiler is explicitly told that he can call f with the intrapage calling convention. So this example will generate the most effective code.

But already this very simple case shows that such a solution is very hard to maintain by hand. Just consider that h must not call f directly, otherwise the code will fail.

Also there are usually not just 3 functions, but thousands or even more. The larger the project, the less this approach is applicable.

Some new linker and compiler features do allow now to optimize complex cases automatically.

This happens in several steps.

1. All functions which should be optimized are put into one distribution segment. As this can be done on a per module, or even on a per application basis with one header file, this does not cause much effort.
2. Then the application is compiled with the conservative assumption that all calls in this segment use the interpage calling convention.
3. The linker is run with this application with the special option -Dist.
   The linker builds a new header file, which assigns a segment for every function to be distributed. The name of this header file can be specified with the option -DistFile.
   Functions only called within the same segment are especially marked. This step does actually build classes of functions which must to be allocated in the same page.

```
...
/* list of all used code segments */

#pragma CODE_SEG __DEFAULT_SEG_CC__ FUNCTIONS0
#pragma CODE_SEG __DEFAULT_SEG_CC__ FUNCTIONS1

/* list of all mapped objects with their calling convention */

#pragma REALLOC_OBJ "FUNCTIONS0" f __NON_INTERSEG_CC__
#pragma REALLOC_OBJ "FUNCTIONS0" g __INTERSEG_CC__
```

```
#pragma REALLOC_OBJ "FUNCTIONS1" h __INTERSEG_CC__
```

The macros __DEFAULT_SEG_CC__, __INTERSEG_CC__ and __NON_INTERSEG_CC__ are set depending on the target processor so that the compiler is using the optimized calling convention, if applicable.

The #pragma CODE_SEG's are defining all used segments, this is a precondition of the "#pragma REALLOC_OBJ". This pragma does then cause the functions to be allocated into the correct segments and tells the compiler when he can use the optimized calling convention.

4. The application is rebuild. This time the linker generated header file is included into every compilation unit.
5. The linker is run again, this time the usual way without the special option. Because of the shorter calling convention used now, some segments will not be completely full. Functions which have the intrasegment calling convention can fill such pages, so that the resulting application does not only runs faster, but also needs less pages.

NOTE Steps 2 to 5 are two usual build processes and can be done with the maker or a batch file.

NOTE As soon as new function calls are added to the sources, all steps from 2 have to be rerun (or the user has to be sure no to call a function with intrapage calling convention across pages).
When the source is modified gets larger, the linking in step 5 may fail. Then steps 2 to 5 have to be repeated.

NOTE The linker does not know whether some functions are called with function pointers.
If this is the case all such functions must be removed from the segment to be optimized in step 1.
This is especially the case for C++ virtual function calls. From the linker’s point of view, a virtual function call is like a function pointer call. So the calling convention of virtual functions cannot be automatically optimized.

New qualifiers and keywords for the optimization:

To determine which sections are banked or not banked it has to be added an IBCC_NEAR (interbank calling convention near) respectively an IBCC_FAR (interbank calling convention far) flag. The distribution segment (in the example down: FUNCTIONS) has to be followed by the "DISTRIBUTE_INTO" keyword (instead of "INTO").

NOTE If you want to use the optimizer don't forget to write "DISTRIBUTE_INTO" instead of "INTO" in the placement of the distribution segment, otherwise the optimizer doesn't work.

Example:

C Source:

```
#pragma CODE_SEG FUNCTIONS
void f(void) { ... }
void g(void) { ... f(); ... }
void h(void) { ... g(); ... }
```

Link parameter File:

```
SECTIONS
...
    MY_ROM0  = READ_ONLY IBCC_NEAR 0x06000 TO 0x07FFF;
    MY_ROM1  = READ_ONLY IBCC_FAR  0x18000 TO 0x18FFF;
    MY_ROM2  = READ_ONLY IBCC_FAR  0x28000 TO 0x28FFF;
...
PLACEMENT
...
    FUNCTIONS DISTRIBUTE_INTO MY_ROM1, MY_ROM2;
...
```

How the optimizer works:

The functions with the most incoming calls and they which are called from outside the distribution segment are inserted into the not banked sections (sections with the "IBCC_Near" flag). Thus they can be called with a near calling convention. The remained functions are arranged like this that in every section will be as few as possible incoming calls (this mean as few as possible calls from a function which is not in the section to an inside one). This can be reached when the caller and the callee are in the same section. A function which is in a banked section (sections with the "IBCC_Far" flag) can only then have a near calling convention if it isn't called by an other function from outside this bank.

Result of the optimization:

With the option -DistInfo an output file can be generated. It contains the result of the optimized distribution. To see the full result of linking it is recommendable to use the -M option to generate a MAPFILE . If necessary it is possible to check which functions from outside of the distribution segment call such from inside. For this the message " Function is not in the distribution segment " has to be enabled which has as default "disabled".

Appropriate options:

Requirements:

The explained method does only work with recent linker version and a compiler supporting the pragma REALLOC_OBJ.

Limitations

There are several points to consider while distributing code in the linker:

- The linker cannot know about calling convention used for function pointers. The compiler can check some simple cases, but in general this is not possible. So be careful while using function pointers that all targets called by function pointers have the correct calling convention set by the memory model. Best is to exclude functions being target of a function pointer call from distribution.
- Actually only one segment can be specified for distribution.
- Usage of HLI: The compiler/linker does not change the HLI code for calling convention. E.g. if a 'far' calling instruction is used in HLI to call a 'near' function, this will not work.
- Linker assumes fixed code sizes for 'far' and 'near' function calling sequences. This is used by the linker to calculate the impact of calling convention change. This way the linker may put some more functions into a segment/bank. However the linker cannot know about other effects of calling convention change.

Checksum Computation

The linker supports two ways how the computation of a checksum can be invoked:

- prm file controlled checksum computation

  The prm file specifies which kind of checksum should be computed over which area and where the resulting checksum should be stored. This method gives the full flexibility, but it also requires more user configuration effort. With this method the linker only computes the actual checksum value. It's up to the

application code to ensure that the area specified in the prm file does match the area computed at runtime.

- automatic linker controlled checksum computation

  With this method, the linker generates a data structure which contains all information to compute the checksum. The linker lists all ROM areas, he computes the checksum and stores them together with area information and type information in a data structure which can then be used at runtime to verify the code.

Table 6.4 Comparison of Checksum methods

Method	Prm file controlled Checksum Computation	Automatic Linker controlled Checksum Computation
Complexity	needs some configuration prm file needs adaptations	easy to use Just call _Checksum_Check
Robustness	values used in the prm file and in the source code have to match. All areas to be checked have to be listed in the prm and the source code.	Good. nothing (or few things) to configure
Control	Everything is in full user control.	Poor. Only if a segment should be checked can be controlled.
Target Memory Usage	Good, only what is needed is present.	Needs more memory because of the control data structure.
Execution time.	mainly depends on method. Too much might be checked as the code size is not exactly known.	mainly depends on method. only needed areas are checked.

Prm file controlled Checksum Computation

The linker can be instructed by some special commands in the prm file to compute the checksum over some explicitly specified areas.

All necessary information for this is specified in the prm file:

Example (in the prm file):

```
CHECKSUM
    CHECKSUM_ENTRY
        METHOD_CRC_CCITT
        OF      READ_ONLY   0xE020 TO 0xFEFF
        INTO    READ_ONLY   0xE010 SIZE 2
        UNDEFINED 0xff
    END
END
```

See the linker command CHECKSUM description for the exact syntax to be used in the prm file and also for more examples.

Automatic Linker controlled Checksum Computation

The linker itself is the one who knows all the memory areas used by an application, therefore this method is using this knowledge to generate a data structure, which then can be used at runtime to validate the complete code.

The linker is providing this information similar to the way it provides copy down and zero out information.

The linker does automatically generate the checksum data structure if the startup data structure has two have additional fields:

```
extern struct _tagStartup {
....
     struct __Checksum* checkSum;
     int nofCheckSums;
....
```

The structure __Checksum is defined in the header file checksum.h:

```
struct __Checksum {
    void* start;
    unsigned int len;
#if _CHECKSUM_CRC_CCITT
    _CheckSum2ByteType checkSumCRC_CCITT;
#endif
#if _CHECKSUM_CRC_16
    _CheckSum2ByteType checkSumCRC16;
#endif
#if _CHECKSUM_CRC_32
```

```
    _CheckSum4ByteType checkSumCRC32;
#endif
#if _CHECKSUM_ADD_BYTE
    _CheckSum1ByteType checkSumByteAdd;
#endif
#if _CHECKSUM_XOR_BYTE
    _CheckSum1ByteType checkSumByteXor;
#endif
};
```

The __checksum structure is allocated by the linker in a ".checksum" section after all the other code or constant sections. As the .checksum section itself must not be checked, it must be the last section in a SECTION list.

The linker is issuing checksum information for all the used segments in the prm file. However, if some segments are filled with a FILL command, then this fill area is not contained.

The checksum types to be computed is derived by the linker by using the field names of the __Checksum structure. Usually only one of the alternatives should be present, but the linker does support to compute any combination checksum methods together.

Automatic struct detection

The linker does read the debug information of the module containing _tagStartup

to detect which checksums it should actually generate and how the structure is built.

Because of this, the structure used by the compiler does always match the structure generated by the linker.

The linker does know the structure field names and the name __Checksum of the checksum structure itself. These names cannot be changed.

The types of the structure fields can be adapted to the actual needs.

.checksum section:

The ".checksum" section must be the last section in a placement. It is allowed to be after the .copy section.

If it is not mentioned in the prm file, its automatically allocated when needed.

The checksum areas do not cover .checksum itself.

Partial Fields

The __Checksum structure can also contain checkSumWordAdd, checkSumLongAdd, checkSumWordXor and checkSumLongXor fields to have checksums computed with larger element sizes. However, as the FILL areas are not considered, the len field might be not a multiple of the element size. When this happens, 0 has to be assumed for the missing bytes. Because this is not handled in the provided example code, automatic generated word or long size add or xor checksums are not officially supported.

Runtime support

The file checksum.h does contain functions prototypes and utilities to compute the various checksums.

The corresponding source file is checksum.c. Check it to find out how to compute the various checksums.

The automatic generated checksum feature does not need any customer code.

A simple call "_Checksum_Check(_startupData.checkSum, _startupData.nofCheckSums);" does state if the checksums are OK.

Linking an Assembly Application

Prm File

When an application consists in assembly files only, the linker prm file can be simplified. In that case:

- No startup structure is required.
- No stack initialization is required, because the stack is directly initialized in the source file.
- No main function is required
- An entry point in the application is required

Example:

```
LINK    test.abs
NAMES   test.o test2.o END
SECTIONS
  DIRECT_RAM = READ_WRITE 0x00000 TO 0x000FF;
  RAM_AREA   = READ_WRITE 0x00300 TO 0x07FFF;
  ROM_AREA   = READ_ONLY  0x08000 TO 0x0FFFF;
PLACEMENT
    myRegister         INTO DIRECT_RAM;
    DEFAULT_RAM        INTO RAM_AREA;
    DEFAULT_ROM        INTO ROM_AREA;
END
INIT Start                    ; Application entry point
VECTOR ADDRESS 0xFFFE Start ; Initialize Reset Vector
```

In the previous example:

- All data sections defined in the assembly input files are allocated in the segment RAM_AREA.
- All code and constant sections defined in the assembly-input files are allocated in the segment ROM_AREA.
- The function MyStart is defines as application entry point and is also specified as reset vector. MyStart must be XDEFed in the assembly source file.

WARNINGS

An assembly application does not need any startup structure or root function.

The two warnings:

`'WARNING: _startupData not found'`

and

`'WARNING: Function main not found'`

can be ignored.

Smart Linking

When an assembly application is linked, smart linking is performed on section level instead of object level. That means that the whole sections containing referenced objects are linked with the application.

Example:

Assembly source file

```
          XDEF entry
dataSec1: SECTION
data1:    DS.W 1
dataSec2: SECTION
data2:    DS.W 2
codeSec:  SECTION
entry:
          NOP
          NOP
          LDX  #data1
          LDD  #5645
          STD  0, X
loop:     BRA loop
```

SmartLinker prm file

```
LINK   test.abs
NAMES  test.o END

SECTIONS
  RAM_AREA   = READ_WRITE 0x00300 TO 0x07FFF;
  ROM_AREA   = READ_ONLY  0x08000 TO 0x0FFFF;
PLACEMENT
    DEFAULT_RAM      INTO RAM_AREA;
    DEFAULT_ROM      INTO ROM_AREA;
END
INIT entry
VECTOR ADDRESS 0xFFE entry
```

In the previous example:

- The function entry is defines as application entry point and is also specified as reset vector.
- The data section ‘dataSec1’ defined in the assembly input file is linked with the application because ‘data1’ is referenced in entry. The section ‘dataSec1’ is allocated in the segment RAM_AREA at address 0x300.
- The code section ‘codeSec’ defined in the assembly-input file is linked with the application because ‘entry’ is the application entry point. The section ‘codeSec’ is allocated in the segment ROM_AREA at address 0x8000.

- The data section 'dataSec2' defined in the assembly input file is not linked with the application, because the symbol 'data2' defined there it is never referenced.

One can choose to switch smart linking OFF for his application. In that case the whole assembly code and objects will be linked with the application.

For the previous example, the prm file used to switch smart linking OFF will look as follows:

ELF Format: (ELF)

```
LINK   test.abs
NAMES  test.o END

SEGMENTS
  RAM_AREA   = READ_WRITE 0x00300 TO 0x07FFF;
  ROM_AREA   = READ_ONLY  0x08000 TO 0x0FFFF;
END
PLACEMENT
    DEFAULT_RAM        INTO RAM_AREA;
    DEFAULT_ROM        INTO ROM_AREA;
END
INIT entry
VECTOR ADDRESS 0xFFE entry
ENTRIES * END
```

HIWARE Format: (HIWARE)

```
LINK   test.abs
NAMES  test.o+ END

SEGMENTS
  RAM_AREA   = READ_WRITE 0x00300 TO 0x07FFF;
  ROM_AREA   = READ_ONLY  0x08000 TO 0x0FFFF;
END
PLACEMENT
    DEFAULT_RAM        INTO RAM_AREA;
    DEFAULT_ROM        INTO ROM_AREA;
END
INIT entry
VECTOR ADDRESS 0xFFFE entry
```

In the previous example:

- The function entry is defines as application entry point and is also specified as reset vector.

- The data section ‘dataSec1’ defined in the assembly input file is allocated in the segment RAM_AREA at address 0x300.
- The data section ‘dataSec2’ defined in the assembly input file is allocated next to the section ‘dataSec1’ at address 0x302.
- The code section ‘codeSec’ defined in the assembly-input file is allocated in the segment ROM_AREA at address 0x8000.

LINK_INFO(ELF)

Some compiler support to write additional information into the ELF file. This information consists out of a topic name and specific content.

```
#pragma LINK_INFO BUILD_NUMBER “12345”
#pragma LINK_INFO BUILD_KIND “DEBUG”
```

The compiler then stores this information into the ELF object file. The linker checks if different object files contain the same topic with a different content. If so, the linker issues a warning.

Finally, the linker issues all LINK_INFO’s into the generated output ELF file.

This feature can be used to warn the user about linking incompatible object files together. Also the debugger can use this feature to pass information from header files used by the compiler into the generated application.

The linker does currently not have any internal knowledge about specific topic names, but it might in the future.

7

The Parameter File

The linker's parameter file is an ASCII text file. For each application you have to write such a file. It contains linker commands specifying how the linking is to be done. This section describes the parameter file in detail, giving examples you may use as templates for your own parameter files. You might also want to take a look at the parameter files of the examples included in your installation version.

The Syntax of the Parameter File

The following is the EBNF syntax of the parameter file.

ParameterFile={Command}.

Command= LINK NameOfABSFile [AS ROM_LIB]

| NAMES ObjFile {ObjFile} END

| SEGMENTS {SegmentDef} END

| PLACEMENT {Placement} END

| (STACKTOP | STACKSIZE) exp

| MAPFILE MapSecSpecList

| ENTRIES EntrySpec {EntrySpec} END

| VECTOR (InitByAddr | InitByNumber)

| INIT FuncName

| MAIN FuncName

| HAS_BANKED_DATA

| OVERLAP_GROUP {FuncName} END

| DEPENDENCY {Dependency} END

| CHECKSUM {ChecksumEntry} END.

NameOfABSFile= FileName.

ObjFile= FileName [”-”].

ObjName= Ident.

QualIden = FileName “:” Ident.

FuncName= ObjName | QualIdent.

MapSecSpecList= MapSecSpec “,” {MapSecSpec}.

EntrySpec= [FileName“:”] (* | ObjName).

MapSecSpec= ALL | NONE | TARGET | FILE | STARTUP | SEC_ALLOC |SORTED_OBJECT_LIST |

OBJ_ALLOC | OBJ_DEP | OBJ_UNUSED | COPYDOWN |

OVERLAP_TREE | STATSTIC.

Dependency= ROOT {ObjName} END

| ObjName USES {ObjName} END

| ObjName ADDUSE {ObjName} END

| ObjName DELUSE {ObjName} END.

SegmentDef= SegmentName “=“ SegmentSpec “;”.

SegmentName= Ident.

SegmentSpec= StorageDevice Range [Alignment] [FILL CharacterList] [OptimizeConstants].

ChecksumEntry= CHECKSUM_ENTRY

ChecksumMethod

[INIT Number]

[POLY Number]

OF MemoryArea

INTO MemoryArea

[UNDEFINED Number]

END.

ChecksumMethod= METHOD_CRC_CCITT | METHOD_CRC8

| METHOD_CRC16 | METHOD_CRC32

| METHOD_ADD | METHOD_XOR.

MemoryArea= StorageDevice Range.

StorageDevice= READ_ONLY | CODE | READ_WRITE | PAGED | NO_INIT.

Range= exp (TO | SIZE) exp.

Alignment= ALIGN [exp] {"["ObjSizeRange":" exp"]"}.

ObjSizeRange= Number | Number TO Number | CompareOp Number.

CompareOp= ("<" | "<=" | ">" | ">=").

CharacterList= HexByte {HexByte}.

OptimizeConstants= {(DO_NOT_OVERLAP_CONSTS |
DO_OVERLAP_CONSTS) {CODE | DATA}}.

Placement= SectionList (INTO | DISTRIBUTE_INTO) SegmentList ";".

SectionList= SectionName {"," SectionName}.

SectionName= Ident.

SegmentList= Segment {"," Segment}.

Segment= SegmentName | SegmentSpec.

InitByAddr= ADDRESS Address Vector.

InitByNumber= VectorNumber Vector.

Address= Number.

VectorNumber= Number.

Vector= (FuncName [OFFSET exp] | exp) ["," exp].

Ident= <any C style identifier>

FileName= <any file name>.

exp= Number.

Number= DecimalNumber | HexNumber | OctalNumber.

HexNumber= 0xHexDigit{HexDigit}.

DecimalNumber= DecimalDigit{DecimalDigit}.

HexByte= HexDigit HexDigit.

HexDigit= "0" | "1"| "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"|
"A" | "B" | "C" | "D" | "E" | "F" |
"a" | "b" | "c" | "d" | "e" | "f" .

DecimalDigit= “0” | “1”| “2” | “3” | “4” | “5” | “6” | “7” | “8” |

“9” |.

Comments may appear anywhere in a parameter file, except where file names are expected. You may use either C style comments or Modula-2 style comments.

File names should not contain paths, this keeps your sources portable. Otherwise, if you copy the sources to some other directory, the linker might not find all files needed. The linker uses the paths in the environment variables GENPATH, OBJPATH, TEXTPATH and ABSPATH to decide where to look for files and where to write the output files.

The order of the commands in the parameter file does not matter. You should only make sure that the SEGMENTS block is specified before the PLACEMENT block.

There are a couple of default sections, named *.data*, *.text*, *.stack*, *.copy*, *.rodata1*, *.rodata*, *.startData*, and *.init*. Information about these sections can be found in chapter predefined sections.

Mandatory SmartLinker Commands

A linker parameter file always has to contain at least the entries for LINK (or using option -O), NAMES, and PLACEMENT. All other commands are optional. The following example shows the minimal parameter file:

```
LINK mini.abs  /* Name of resulting ABS file */
NAMES
  mini.o startup.o  /* Files to link */
END
STACKSIZE 0x20       /* in bytes */
PLACEMENT
  DEFAULT_ROM INTO READ_ONLY  0xA00 TO 0xBFF;
  DEFAULT_RAM INTO READ_WRITE 0x800 TO 0x8FF;
END
```

In case the linker is called by CodeWarrior, then the LINK command is not necessary. The CodeWarrior Plug-In passes the option -O with the destination file name directly to the linker. You can see this if you enable ‘Display generated command lines in message window’ in the Linker preference panel in CodeWarrior.

The first placement statement

```
DEFAULT_ROM INTO READ_ONLY 0xA00 TO 0xBFF;
```

reserves the address range from 0xA00 to 0xBFF for allocation of read only objects (hence the qualifier READ_ONLY). .text subsumes all linked functions, all constant variables, all string constants and all initialization parts of variables, copied to RAM at startup.

The second placement statement

```
DEFAULT_RAM INTO READ_WRITE 0x800 TO 0x8FF;
```

reserves the address range from 0x800 to 0x8FF for allocation of variables.

The INCLUDE directive

A special directive INCLUDE allows to split up a prm file into several text files, for example to separate a target specific part of a prm file from a common part.

The syntax of the include directive is:

```
IncludeDir= "INCLUDE" FileName.
```

Because the INCLUDE directive may be everywhere in the prm file, it is not contained in the main EBNF.

Example:

```
LINK mini.abs  /* Name of resulting ABS file */
NAMES
  startup.o  /* startup object file */
  INCLUDE objlist.txt
END
STACKSIZE 0x20       /* in bytes */
PLACEMENT
  DEFAULT_ROM INTO READ_ONLY  0xA00 TO 0xBFF;
  DEFAULT_RAM INTO READ_WRITE 0x800 TO 0x8FF;
END
with objlist.txt:
  mini0.o  /* user object file(s) */
  mini1.o
```

8

SmartLinker Commands

This section describes the details of each linker parameter command.

Each command has at least following description:

- **Syntax**: Description of the command syntax.
- **Description**: Detailed description of the command.
- **Example**: Example how to use the command.

Some commands are only available in ELF/Dwarf format, and some commands only in HIWARE object file format. This marked with the object file format in parenthesis (ELF) or (HIWARE).

If a command is only available for a specific language, it is marked too, for example, ‘M2’ denotes that the feature is only available for Modula-2 linker parameter files.

Additionally, if the behavior of a command is different in HIWARE or in ELF/Dwarf format, this is mentioned too.

AUTO_LOAD

AUTO_LOAD: Load Imported Modules (HIWARE, M2)

Syntax

```
AUTOLOAD ON | OFF
```

Description:

AUTO_LOAD is an optional command having an effect on linking only when there are Modula2 modules present. When AUTO_LOAD is switched ON, the linker automatically loads and processes all modules imported in some Modula2 module, i.e.

it is not necessary to enumerate all object files of Modula-2applications. The linker assumes that the object file name of a Modula-2 module is the same as the module name with extension."o". Modules loaded by the linker automatically (i.e. imported in some Modula-2 Module present in the NAMES list) must not appear in the NAMES list. The default Setting is ON.

AUTO_LOAD must be switched OFF when linking with a ROM library. If it is switched ON, the linker would automatically load the missing object files, thus disregarding the objects in the ROM library.

Note: AUTO_LOAD must also be switched OFF if the object file names are not the same as the module names, because in this case the linker is unable to find the object files.

Example:

```
AUTOLOAD ON
```

CHECKSUM

CHECKSUM: Checksum computation (ELF)

Syntax

Checksum= CHECKSUM {ChecksumEntry} END.

ChecksumEntry= CHECKSUM_ENTRY

ChecksumMethod

[INIT Number]

[POLY Number]

OF MemoryArea

INTO MemoryArea

[UNDEFINED Number]

END.

ChecksumMethod= METHOD_CRC_CCITT | METHOD_CRC8

| METHOD_CRC16 | METHOD_CRC32

| METHOD_ADD | METHOD_XOR.

Description:

The linker can be instructed with this directives to compute the checksum over some memory areas.

All necessary information for this is specified in this structure.

Note that the OF MemoryArea specified usually also has its separate SEGMENTS entry. It is recommended to use the FILL directive there to actually fill all gaps to get a predictable result.
E.g.:

```
SEGMENTS
MY_ROM = READ_ONLY   0xE020 TO 0xFEFF FILL 0xFF;
....
END
CHECKSUM
  CHECKSUM_ENTRY METHOD_CRC_CCITT
    OF      READ_ONLY   0xE020 TO 0xFEFF
    INTO    READ_ONLY   0xE010 SIZE 2
    UNDEFINED 0xff
  END
END
```

The checksum can only computed over areas with READ_ONLY and CODE qualifiers.

The following methods are supported:

- METHOD_XOR. The elements of the memory area are xored together.
  The element size is defined by the size of the INTO_AREA.
- METHOD_ADD. The elements of the memory area are added together.
  The element size is defined by the size of the INTO_AREA.
- METHOD_CRC_CCITT. A 16-bit CRC (cyclic redundancy check) checksum according to CRC CCITT is computed over all bytes in the area. The INTO_AREA size must be 2 bytes.
- METHOD_CRC16. A 16-bit CRC checksum according to the commonly used CRC 16 is computed over all bytes in the area. The INTO_AREA size must be 2 bytes.

- METHOD_CRC32. A 32-bit CRC checksum according to the commonly used CRC 32 is computed over all bytes in the area. The INTO_AREA size must be 4 bytes.

The optional [INIT Number] entry is used as initial value in the checksum computation. If it is not specified, a default values of 0xffffffff for CRC checksums and 0 for addition and xor is used.

The optional [POLY Number] entry allows to specify alternative polynomials for the CRC checksum computation.

OF MemoryArea: The area of which the checksum should be computed.

INTO MemoryArea: The area into which the computed checksum should be stored. It be distinct from any other placement in the prm file and from the OF MemoryArea.

The optional [UNDEFINED Number] value is used when no memory is at certain places. However it is recommended to use the FILL directive to avoid this (for an example see above).

Example 1:

```
CHECKSUM
   CHECKSUM_ENTRY
      METHOD_CRC_CCITT
      OF      READ_ONLY   0xE020 TO 0xFEFF
      INTO    READ_ONLY   0xE010 SIZE 2
      UNDEFINED 0xff
   END
END
```

This entry causes the computation of a checksum from 0xE020 up to 0xFEFF (including this address).

The checksum is calculated according to the CRC CCITT.

Example 2:

Example:

Assume the following memory content:

0x1000 02 02 03 04

Then the XOR 1 byte checksum from 0x1000 to 0x1003 is 0x06 (=0x02^0x02^0x03^0x04).

Notes:

- METHOD_XOR is the fastest method to compute together with METHOD_ADD.

- However, for METHOD_XOR and METHOD_ADD, multiple regular one bit changes can cancel each other out. The CRC methods avoid this weakness.

As example, assume that both 0x1000 and 0x1001 are getting cleared, then, the XOR checksum does not change. There are similar cases for the addition as well.

- METHOD_XOR/METHOD_ADD do also support to compute the checksum with larger element sizes.

The element size is taken as the size of the INTO MemoryArea part.

With a element size of 2, the checksum of the example would be 0x0506 (= 0x2020 ^ 0x0304).

Larger element sizes do allow a faster computation of the checksums on 16 or 32 bit machines.

The size and the address of the OF MemoryArea part have to be a multiple of the element size.

CRC checksums do only compute the values byte wise (or more precisely they are even defined bitwise).

- Often, the actual size of the area to be checked is not known in advance.

Depending on how much code the compiler is generating for C source code, the placements do fill up more or less.

This method however does not support varying sizes. Instead, the unused areas in the placement have to be filled with the FILL directive to a known value. This causes a certain overhead as the checksum is computed over these fill areas as well.

CHECKKEYS

CHECKKEYS: Check Module Keys (HIWARE, M2)

Syntax

```
CHECKKEYS ON | OFF
```

Description:

The CHECKKEYS command is optional. If switched ON (which is the default), the linker compares module keys of the Modula-2 modules in the application and issues an error message if there is an inconsistency (symbol file newer than the object file). CHECKKEYS OFF turns off this module key check.

Example:

```
CHECKKEYS ON
```

DATA

DATA: Specify the RAM Start (HIWARE)

Syntax

```
DATA  Address
```

Description

This is a command supported in ‘old-style’ linker parameter files and will be not supported in a future release.

With this command the default ROM begin can be specified. The specified address has to be in hexadecimal notation. Internally this command is translated into

DATA 0x????' => 'DEFAULT_RAM INTO READ_WRITE 0x???? TO 0x????

Note that because the end address is of DEFAULT_RAM is not known, the linker tries to specify/find out the end address itself. Because this is not a very transparent behavior, this command will not be supported any more.

Example

```
START 0x1000
```

DEPENDENCY

DEPENDENCY: Dependency Control

Syntax

```
DEPENDENCY {Dependency} END.

Dependency = ROOT {ObjName} END

| ObjName USES {ObjName} END

| ObjName ADDUSE {ObjName} END

| ObjName DELUSE {ObjName} END.
```

Description

The keyword DEPENDENCY allows the modification of the automatically detected dependency information.

New roots can be added (ROOT keyword) and existing dependencies can be overwritten (USES), extended (ADDUSE) or removed (DELUSE).

The dependency information is mainly used for 2 purposes:

- Smart Linking Only objects depending on some roots are linked at all.
- Overlapping of local variables and parameters
  Some small 8 bit processors are using global memory instead of stack space to allocate local variables and parameters. The linker uses the dependency

information to allocate local variables of different functions which never are active at the same time to the same addresses.

ROOT

With the ROOT keyword a group of root objects can be specified.

A ROOT entry with a single object is semantically the same as if the object would be in a ENTRIES section. A ROOT entry with several objects is semantically the same as an OVERLAP_GROUP entry (which is however only available in ELF). If several objects however are in one root group, there is an additional semantic that only one object of the group is active at the same time. This information is used for an improved overlapped allocation of variables. Variables of functions of the same group are allocated in the same area. If you do not want to specify this, either use several ROOT blocks or add the objects in the ENTRIES section.

Example (Overlapped allocation of variables, only for some targets):

C source:

```
void main(void) { int i; ....}
void interrupt int1(void) { int j; ... }
void interrupt int2(void) { int k; ... }
```

prm file:

```
...DEPENDENCY
  ROOT main END
  ROOT int1 int2 END
END
```

In this example, the variables of the function main and all its dependents are allocated first. Then the variables of int1 and int2 are allocated into the same area. So j and k may overlap.

USES

The USES keyword defines all dependencies for a single object. Only the given dependencies are used. Any not listed dependencies are not taken into account. If a needed dependency is not specified after the USES, the linker will complain.

Example (Overlapped allocation of variables, only for some targets)

C Source:

```
void f(void(* fct)(void)) { int i; ... fct();...}
void g(void) { int j;... }
void h(void) { int k;... }
void main(void) { f(g); f(h); }
```

prm file:

```
DEPENDENCY
  f USES g h END
END
```

This USES statement does assure that the variable i of f does not overlap any of the variables of g or h.

The automatic detection does not work for functions called by a function pointer initialized outside of the function as in this case.

However the USES keyword hides any dependencies specified by the compiler. Only if the code of f not shown above does not call additional functions, this USES is safe. It is usually better to use ADDUSE, explained below, than to use USES.

ADDUSE

The ADDUSE keyword allows to add additional dependencies to the ones automatically detected. The ADDUSE is safe in the way that no dependencies are lost. So the generated application might use more memory than necessary, but it does consider all known dependencies.

Example (Overlapped allocation of variables, only for some targets)

C Source:

```
void f(void(* fct)(void)) { int i; ... fct();...}
void g(void) { int j;... }
void h(void) { int k;... }
void main(void) { f(g); f(h); }
```

prm file:

```
DEPENDENCY
  f ADDUSE g h END
END
```

This example is safer than the pervious version with USES because only new dependencies are added.

For smart linking, the automatic detection covers almost all cases. Only if some objects are accessed by a fix address, for example, one must link additional depending objects.

Example (Smart Linking)

C-Code:

```
int i @ 0x8000;
void main(void) {
  *(int*)0x8000 = 3;
}
```

To tell the linker that i has to be linked too, if main is linked, the following line can be added to the link parameter file:

`DEPENDENCY main ADDUSE i END`

DELUSE

The DELUSE keyword allows to remove single dependencies from the set of automatic detected dependencies.

To get a list of all automatic detected dependencies, comment out any DEPENDECY block in the prm file, switch on the map file generation and see the "OBJECT-DEPENDENCIES SECTION" in the generated map file.

The automatic generation of dependencies can generate unnecessary dependencies because, for example, the runtime behavior is not taken into account.

Example

C Source:

```
void MainWaitLoop(void) { int i; for (;;) { ... } }
```

```
void _Startup(void) { int j; InitAll();
 MainWaitLoop(void); }
```

prm file:

```
DEPENDENCY
  _Startup DELUSE MainWaitLoop END
  ROOT _Startup MainWaitLoop END
END
```

Because MainWaitLoop does not take any parameter and does never return, its local variable i can be allocated overlapped with _Startup. The ROOT directive specifies that the locals of the two functions can be allocated at the same addresses.

Overlapping of local variables and parameters

The most common application of the DEPENDENCY command is for the overlapping.

See Also

Keyword OVERLAP_GOURP
Overlapping Locals

ENTRIES

ENTRIES: List of Objects to Link with the Application

Syntax (ELF):

```
ENTRIES
   [FileName " :"](*|objName)
   {[FileName ":"](*|objName)}
END
```

Syntax (HIWARE):

```
ENTRIES objName {objName} END
```

Description

The ENTRIES block is optional in a prm file and it cannot be specified several times.

The ENTRIES block is used to specify a list of objects, which must always be linked with the application, even when they are never referenced. The specified objects are used as additional entry point in the application. That means all objects referenced within these objects will also be linked with the application.

Table 8.1 describes the notation that are supported.

Table 8.1 Notation and their description

Notation	Description
<Object Name>	The specified global object must be linked with the application
<File Name>:<Object Name> (ELF)	The specified local object defined in the specified binary file must be linked with the application

Table 8.1 Notation and their description

Notation	Description
<File Name>:* (ELF)	All objects defined within the specified file must be linked with the application
* (ELF)	All objects must be linked with the application. This switches OFF smart linking for the application

ELF Specific issues (ELF):

If a file name specified in the ENTRIES block is not present in the NAMES block, this file name is inserted in the list of binary files building the application.

Example

```
NAMES
  startup.o
END

ENTRIES
  fibo.o:*
END
```

In the previous example, the application is build from the files fibo.o and startup.o.

File Names specified in the ENTRIES block may also be present in the NAMES block.

Example

```
NAMES
  fibo.o startup.o
END

ENTRIES
  fibo.o:*
END
```

In the previous example, the application is build from the files fibo.o and startup.o. The file 'fibo.o' specified in the ENTRIES block is the same as the one specified in the ENTRIES block.

NOTE We strongly recommend to avoid switching smart linking OFF, when the ANSI library is linked with the application. The ANSI library contains the implementation of all run time functions and ANSI standard functions. This generates a large amount of code, which is not required by the application.

HAS_BANKED_DATA

HAS_BANKED_DATA: Application has banked data (HIWARE)

Syntax

```
HAS_BANKED_DATA
```

Description

This entry is used to specify for the HC12 in the HIWARE object file format that all pointers in the zero out and in the copy down must be 24 bit in size.

In the ELF object file format, this entry is ignored.

Example

```
HAS_BANKED_DATA
```

HEXFILE

HEXFILE: Link a Hex File with the Application

Syntax

```
HEXFILE <fileName> [OFFSET <hexNumber>]
```

Arguments

<fileName> is any valid file name. This file is searched in the current directory first, and then in the directories specified in the environment variable "GENPATH".

<hexNumber> if specified, this number is added to the address found in each record of the hex file. The result is then the address where the data bytes are copied to.

Description

Using this command a Motorola S-Record file or a Intel Hex file can be linked with the application.

Example:

```
HEXFILE fiboram.s1 OFFSET 0xFFFF9800 /* 0x800 - 0x7000 */
```

The optional offset specified in the HEXFILE command is added to each record in the Motorola S file. The code at address 0x7000 will be encoded at address 0x800. The offset 0xFFFF9800 used above is the unsigned representation of -0x68000. To calculate it, use a hex capable calculator, for example the Windows Calculator in scientific mode, and subtract 0x7000 from 0x800.

NOTE	Be careful, in the HIWARE Format, no checking is performed to avoid overwriting of any portion of normal linked code by data from hex files.

Example

```
HEXFILE fiboram.s1 OFFSET 0xFFFF9800 /* 0x800 - 0x7000 */
```

INIT

INIT: Specify the Application Init Point

Syntax

```
INIT  FuncName
```

Description

The INIT command is mandatory for assembly application and optional otherwise. It cannot be specified several times in the prm file. This command defines the initialization entry point for the application used.

When INIT is not specified in the prm file, the linker looks for a function named '_Startup' and use it as application entry point.

If an INIT command is specified in the prm file, the linker uses the specified function as application entry point. This is either the main routine or a startup routine calling the main routine.

ELF Specific issues (ELF):

You can specify any static or global function as entry point.

Example

```
INIT MyGlobStart /* Specify a global variable as application
entry point.*/
```

ELF Specific Example (ELF):

```
INIT myFile.o:myLocStart /* Specify a local variable
as application entry point.*/
```

This command is not used for ROM libraries. If you specify an INIT command in a ROM library prm file, a warning is generated.

LINK

LINK: Specify Name of Output File

Syntax

```
LINK <NameOfABSFile> ['AS ROM_LIB']
```

Description

The LINK command defines the name of the file which should be generated by the link session. This command is mandatory and can only be specified once in a prm file.

After a successful link session the file “`NameOfABSFile`” is created. If the environment variable ABSPATH is defined, the absolute file is generated in the first directory listed there. Otherwise, it is written to the directory where the parameter file was found. If a file with this name already exists, it is overwritten.

A successful linking session also creates a map file with the same base name as “`NameOfABSFile`” and with extension .map. If the environment variable TEXTPATH is defined, the map file is generated in the first directory listed there. Otherwise, it is written to the directory where the parameter file was found. If a file with this name already exists, it is overwritten.

If the name of the absolute file is followed by AS ROM_LIB, a so-called ROM library is generated instead of an absolute file (Please see section *ROM Libraries*). A ROM library is an absolute file which is not executable alone.

The LINK command is mandatory in a prm file. If the LINK command is missing the SmartLinker generates an error message unless the option -O is specified on the command line. Note that if the Linker is started from CodeWarrior, the option -O is automatically added.

If the option -O is specified on the command line, option -O has higher priority than LINK command.

Example

```
LINK fibo.abs

NAMES  fibo.o startup.o END
SECTIONS
    MY_RAM = READ_WRITE 0x1000 TO 0x18FF;
    MY_ROM = READ_ONLY  0x8000 TO 0x8FFF;
    MY_STK = READ_WRITE 0x1900 TO 0x1FFF;
PLACEMENT
    DEFAULT_ROM    INTO  MY_ROM;
    DEFAULT_RAM    INTO  MY_RAM;
    SSTACK         INTO  MY_STK;
END
VECTOR ADDRESS 0xFFFE _Startup /* set reset vector */
```

The files fibo.ABS and a fibo.map are generated after successful linking from the previous prm file.

MAIN

MAIN: Name of the Application Root Function

Syntax

```
MAIN  FuncName
```

Description

The MAIN command is optional and cannot be specified several times in the prm file. This command defines the root function for an ANSI C application (the function which is invoked at the end of the startup function.

When MAIN is not specified in the prm file, the linker looks for a function named ‘main’ and use it as application root.

If a MAIN command is specified in the prm file, the linker uses the specified function as application root.

ELF Specific issues (ELF):

You can specify any static or global function as application root function.

Example

```
MAIN MyGlobMain /* Specify a global variable as
application root.*/
```

ELF Specific Example (ELF):

```
MAIN myFile.o:myLocMain /* Specify a local variable as
application root.*/
```

This command is not required for ROM libraries. If you specify a MAIN command in a ROM library prm file, a warning is generated.

MAPFILE

MAPFILE: Configure Map File Content

Syntax (ELF):

```
MAPFILE (ALL|NONE|TARGET|FILE|STARTUP_STRUCT|SEC_ALLOC|
    OBJ_ALLOC|SORTED_OBJECT_LIST|OBJ_DEP|OBJ_UNUSED|
```

```
    COPYDOWN|OVERLAP_TREE|STATISTIC|MODULE_STATISTIC)
[,{(ALL|NONE|TARGET|FILE|STARTUP_STRUCT|SEC_ALLOC|OBJ_ALLOC
|OBJ_DEP|OBJ_UNUSED|COPYDOWN|OVERLAP_TREE|STATISTIC|MODULE_STA
    TISTIC)}]
```

Syntax (HIWARE):

```
MAPFILE (ON|OFF)
```

Description

This command is optional, it is used to control the generation of the Map file. Per default, the command MAPFILE ALL is activated, indicating that a map file must be created, containing all linking time information.

Table 8.2 describes the map file specifiers that are available.

Table 8.2 Map file specifiers and their description

Specifier	Description
ALL (ELF)	A map file must be generated containing all information available
COPYDOWN(ELF)	The information about the initialization value for objects allocated in RAM must be written to the map file (Section COPYDOWN in the map file)
FILE(ELF)	The information about the files building the application must be inserted in the map file (Section FILE in the map file).
NONE(ELF)	No map file must be generated
OBJ_ALLOC(ELF)	The information about the allocated objects must be inserted in the map file (Section OBJECT ALLOCATION in the map file)
SORTED_OBJECT_LIST(ELF)	The map file must contain a list of all allocated objects sorted by the address. (Section OBJECT LIST SORTED BY ADDRESS in the map file)
OBJ_UNUSED(ELF)	The list of all unused objects must be inserted in the map file (Section UNUSED OBJECTS in the map file)

Table 8.2 Map file specifiers and their description (*continued*)

Specifier	Description
OBJ_DEP(ELF)	The dependencies between the objects in the application must be inserted in the map file (Section OBJECT DEPENDENCY in the map file).
DEPENDENCY_TREE(ELF)	The dependency tree shows how the overlapped variables are allocated (Section DEPENDENCY TREE in the map file).
OFF (HIWARE)	No map file must be generated
ON (HIWARE)	A map file must be generated containing all information available
SEC_ALLOC(ELF)	The information about the sections used in the application must be inserted in the map file (Section SECTION ALLOCATION in the map file)
STARTUP_STRUCT(ELF)	The information about the startup structure must be inserted in the map file (Section STARTUP in the map file).
MODULE_STATISTIC(ELF)	The MODULE STATISTICS tell how much ROM/RAM is used by a specific module (module is used here as synonym for compilation unit).
STATISTIC(ELF)	The statistic information about the link session must be inserted in the map file (Section STATISTICS in the map file)
TARGET(ELF)	The information about the target processor and memory model must be inserted in the map file (Section TARGET in the map file).

The kind of information generated for each specifier is described latter on in chapter map file.

ELF Specific issues (ELF):

As soon as ALL is specified in the MAPFILE command, all sections are inserted in the map file.

Example

Following commands are all equivalents. A map file is generated, which contains all the possible information about the linking session.

```
MAPFILE ALL
MAPFILE TARGET, ALL
MAPFILE TARGET, ALL, FILE, STATISTIC
```

As soon as NONE is specified in the MAPFILE command, no map file is generated.

Example

Following commands are all equivalents. No map file is generated.

```
MAPFILE NONE
MAPFILE TARGET, NONE
MAPFILE TARGET, NONE, FILE, STATISTIC
```

NOTE For compatibility with old style HIWARE format prm file, following commands are also supported:
MAPFILE OFF is equivalent to MAPFILE NONE
MAPFILE ON is equivalent to MAPFILE ALL

NAMES

NAMES: List the Files building the Application.

Syntax

```
NAMES <FileName>['+'|'-'] {<FileName>['+'|'-']} END
```

Description

The NAMES block contains a list of binary files building the application. This block is mandatory and can only be specified once in a prm file.

The linker reads all files given between NAMES and END. The files are searched for first in the project directory, then in the directories specified in the environment variable OBJPATH and finally in the directories specified in the environment variable GENPATH. The files may be either object files, absolute or ROM Library files or libraries.

Additional files may be specified by the option -Add. The object files specified with the option -Add are linked before the files mentioned in the NAMES block.

As the SmartLinker is a smart linker, only the referenced objects (variables and functions) are linked to the application. You can specify any number of files in the NAMES block, because of smart linking, the application only contains the functions and variables really used.

The plus sign after a file name (e.g. `FileName+`) switches OFF smart linking for the specified file. That means, all the objects defined in this file will be linked with the application, regardless whether they are used or not.

A minus sign can also be specified after an absolute file name (e.g. `FileName-`). This indicates that the absolute file should not be involved in the application startup (global variables defined in the absolute file should not be initialized during application startup) (Please see section *Using ROM Libraries*).

No blank is allowed between the file name and the plus or minus sign.

Example

```
LINK fibo.abs

NAMES  fibo.o startup.o END
SEGMENTS
    MY_RAM = READ_WRITE 0x1000 TO 0x18FF;
    MY_ROM = READ_ONLY  0x8000 TO 0x8FFF;
    MY_STK = READ_WRITE 0x1900 TO 0x1FFF;
PLACEMENT
    DEFAULT_ROM    INTO  MY_ROM;
    DEFAULT_RAM    INTO  MY_RAM;
    SSTACK         INTO  MY_STK;
END
VECTOR ADDRESS 0xFFFE _Startup /* set reset vector */
```

In the previous example, the application fibo is build from the files ‘fibo.o’ and ‘startup.o’.

OVERLAP_GROUP

OVERLAP_GROUP: Application uses Overlapping (ELF)

Syntax

```
OVERLAP_GROUP {<Objects>} END
```

Description

The OVERLAP_GROUP is used for overlapping of locals only. See also the chapter Overlapping Locals.

In some cases the linker cannot detect that there is no dependency between some functions, so that local variables are not overlapped, even if this would be possible. A OVERLAP_GROUP block allow the user to specify a group of functions, which does not overlap.

OVERLAP_GROUP is only available in the ELF object file format. However, the same functionality can be achieved with the DEPENDENCY ROOT command, which is also available in the HIWARE format.

Example:

Assume the default implementations of the C startup routines:

- _Startup: the main entry point of the application. It calls first Init and then uses _startupData to call main.
- Init: Uses the information in _startupData to generate the zero out
- _startupData: Data-structure filled by the linker containing various information as the address of the main function and which areas are to be handled by the zero out in Init.
- main: The main startup point of C code

Between these objects, the following dependencies exist:

- _Startup depends on _startupData, Init
- Init depends on _startupData
- _startupData depends on main.

Assume the following entry in the prm file:

```
/* _Startup is a group of it's own */
OVERLAP_GROUP _Startup END
```

When investigating _Startup, linker does not know that Init does not call main. According to the dependency information, it might call main, so the variables of Init and main are not overlapped.

But in this case, the following OVERLAP_GROUP is build in the linker:

```
/* Overlap the variables of main and the variables of _Startup */
OVERLAP_GROUP main _Startup END
```

This way, the linker overlaps the variables of Init and main because first main is allocated and then _Startup.

For the HC05 with the usual startup code, this entry saves 8 bytes in the _OVERLAP segment. But if the usual startup code is modified the way that _Startup and main must not overlap, insert "OVERLAP_GROUP _Startup END" into the prm file.

NOTE All the name of the _Startup function, of main and of _startupData can be configured in prm file to a non-default one.

Example:

Assume that a processor has two interrupt priorities.

Assume two functions IntPrio1A and IntPrio1B handle interrupt 1 priority requests and the two functions IntPrio0A and IntPrio0B handle the interrupt 0 priority requests. As never two function on the same priority level can be active at the same time, two OVERLAP_GROUPs can be used to overlap the functions of the same level.

```
OVERLAP_GROUP IntPrio1A IntPrio1B END
OVERLAP_GROUP IntPrio0A IntPrio0B END
```

See also

keyword DEPENDENCY

Overlapping Locals

PLACEMENT

PLACEMENT: Place Sections into Segments

Syntax

```
PLACEMENT
  SectionName{,sectionName} (INTO | DISTRIBUTE_INTO)
  SegSpec{,SegSpec};
  {SectionName{,sectionName} (INTO | DISTRIBUTE_INTO)
  SegSpec{,SegSpec};}
END
```

Description

The PLACEMENT block is mandatory in a prm file and it cannot be specified several times.

Each placement statement between the PLACEMENT and END defines a relation between logical sections and physical memory ranges called segments.

Example

```
SECTIONS
  ROM_1  = READ_ONLY 0x800 TO 0xAFF;
PLACEMENT
  DEFAULT_ROM, ROM_VAR INTO ROM_1;
END
```

In the previous example, the objects from section ‘DEFAULT_ROM’ are allocated first and then the objects from section ‘ROM_VAR’.

Allocation of the objects starts with the first section in the list; they are allocated in the first memory range in the list as long as there is enough memory left. If a segment is

full (i.e. the next object to be allocated doesn't fit anymore), allocation continues in the next segment in the list.

Example

```
SEGMENTS
  ROM_1  = READ_ONLY 0x800 TO 0xAFF;
  ROM_2  = READ_ONLY 0xB00 TO 0xCFF;
END
PLACEMENT
  DEFAULT_ROM INTO ROM_1, ROM_2;
END
```

In the previous example, the objects from section 'DEFAULT_ROM' are allocated first in segment 'ROM_1'. As soon as the segment 'ROM_1' is full, allocation continues in section 'ROM_2'.

A statement inside of the PLACEMENT block can be split over several lines. The statement is terminated as soon as a semicolon is detected.

The SECTIONS block must always be defined in front of the PLACEMENT block, because the segments referenced in the PLACEMENT block must previously be defined in the SECTIONS block.

Some restrictions applies on the commands specified in the PLACEMENT block:

- When the .copy section is specified in the PLACEMENT block, it should be the last section in the section list.
- When the .stack section is specified in the PLACEMENT block, an additional STACKSIZE command is required in the prm file, when the stack is not the single section specified in the placement statement.
- The predefined sections .text and .data must always be specified in the PLACEMENT block. They are used to retrieve the default placement for code or variable sections. All code or constant sections, which do not appear in the PLACEMENT block are allocated in the same segment list as the .text section. All variable sections, which do not appear in the PLACEMENT block are allocated in the same segment list as the .data section.

PRESTART

PRESTART: Application Prestart Code (HIWARE)

Syntax

```
PRESTART (["+"] HexDigit {HexDigit} | OFF)
```

Description

This is an optional command. It allows the modification of the default init code generated by the linker at the very beginning of the application. Normally this code looks like

```
DisableInterrupts.
On some processor, setup page registers
JMP StartupRoutine ("_Startup" by default)
```

If a PRESTART command is given, all code before the JMP is replaced by the code given by the Hex numbers following the keyword. If there is a "+" following the PRESTART, the code given does not replace the standard sequence but is inserted just before JMP.

Note: After the PRESTART command do not write a sequence of hexadecimal numbers in C (or Modula-2) format! Just write an even number of hexadecimal digits. Example:

```
PRESTART + 4E714E71
```

PRESTART OFF turns off prestart code completely, i.e. the first instruction executed is the first instruction of the startup routine.

Example

```
PRESTART OFF
```

SECTIONS

SECTIONS: Define Memory Map

Syntax

```
SECTIONS {(READ_ONLY|READ_WRITE|NO_INIT|PAGED)
           <startAddr> (TO <endAddr> | SIZE <size>)}
```

Description

The **SECTIONS** block is optional in a prm file and it cannot be specified several times. The **SECTIONS** block must be directly followed by the **PLACEMENT** block.

The **SECTIONS** command allows the user to assign meaningful names to address ranges. These names can then be used in subsequent placement statements, thus increasing the readability of the parameter file.

Each address range you define is associated with

- a qualifier.
- a start and end address or a start address and a size.

Section Qualifier

Following qualifier are available for sections:

- READ_ONLY: used for address ranges, which are initialized at program load time. The application (*.abs) does only contain content for this qualifier.
- READ_WRITE: used for address ranges, which are initialized by the startup code at runtime. Memory area defined with this qualifier will be initialized with 0 at application startup. The information how the READ_WRITE section is initialized is stored in a READ_ONLY section.
- NO_INIT: used for address ranges, where read write accesses are allowed. Memory area defined with this qualifier will not be initialized with 0 at

application startup. This may be useful if your target has a battery buffered RAM or to speedup application startup.

- PAGED: used for address ranges, where read write accesses are allowed. Memory area defined with this qualifier will not be initialized with 0 at application startup. Additionally, the linker will not control if there is an overlap between segments defined with the PAGED qualifier. When overlapped segments are used, it is the user's responsibility to select the correct page before accessing the data allocated on a certain page.

Qualifier Handling

Qualifier	Initialized Variables	Non-Initialized Variables	Constants	Code
READ_ONLY	not applicable (1)	not applicable (1)	content written to target address	content written to target address
READ_WRITE	content written into copy down area, together with info where to copy it at startup. Area contained in zero out information (3) (4)	Area contained in zero out information (4)	content written into copy down area, together with info where to copy it at startup. Area contained in zero out information (3) (4)	not applicable (1) (2)
NO_INIT	not applicable (1)	just handled as allocated. Nothing generated.	not applicable (1)	not applicable (1)
PAGED	not applicable (1)	just handled as allocated. Nothing generated.	not applicable (1)	not applicable (1)

1. These cases are not intended. The linker does however allow some of them. If so, the qualifier controls what is written into the application.
2. To allocate code in a RAM area, for example for testing purposes, declare this area as READ_ONLY.

3. Initialized objects and constant in READ_WRITE sections do need additionally to the RAM memory, also space in the copy down area. The copy down contains the information how the object is initialized in the startup code.
4. The zero out information consist of the information which areas should be initialized with 0 at startup. Because the zero out contains only an address and a size per area, it is usually much smaller than a copy down area, which also contains the (non zero) content of the objects to be initialized.

Example

```
SECTIONS
  ROM   = READ_ONLY  0x1000 SIZE 0x2000;
  CLOCK = NO_INIT    0xFF00 TO   0xFFFF;
  RAM   = READ_WRITE 0x3000 TO   0x3EFF;
  Page0 = PAGED      0x4000 TO   0x4FFF;
  Page1 = PAGED      0x4000 TO   0x4FFF;
END
```

In the previous example:

- the segment ’ROM’ is a READ_ONLY memory area. It starts at address 0x1000 and its size is 0x2000 bytes (from address 0x1000 to 0x2FFF).
- The segment ’RAM’ is a READ_WRITE memory area. It starts at address 0x3000 and ends at 0x3FFF (size = 0x1000 bytes). All variables allocated in this segment will be initialized with 0 at application startup.
- The segment ’CLOCK’ is a READ_WRITE memory area. It starts at address 0xFF00 and ends at 0xFFFF (size = 100 bytes). Variables allocated in this segment will not be initialized at application startup.
- The segments ’Page0’ and ‘Page1’ is a READ_WRITE memory area. These are overlapping segments. It is the user responsibility to select the correct page before accessing any data allocated in one of these segment. Variables allocated in this segment will not be initialized at application startup.

SEGMENTS

SEGMENTS: Define Memory Map (ELF)

Syntax

```
SEGMENTS {(READ_ONLY|READ_WRITE|NO_INIT|PAGED)
            <startAddr> (TO <endAddr> | SIZE <size>)
           [ALIGN <alignmentRule>] [FILL <fillPattern>]
           {(DO_OPTIMIZE_CONSTS | DO_NOT_OPTIMIZE_CONSTS)
              { CODE | DATA }
           }
          }
END
```

Description

The **SEGMENTS** block is optional in a prm file and it cannot be specified several times.

The **SEGMENTS** command allows the user to assign meaningful names to address ranges. These names can then be used in subsequent placement statements, thus increasing the readability of the parameter file.

Each address range you define is associated with:

- a qualifier.
- a start and end address or a start address and a size.
- an optional alignment rule
- an optional fill pattern.
- optional constant optimization with Common Code commands.

Segment Qualifier

Following qualifier are available for segments:

- READ_ONLY: used for address ranges, which are initialized at program load time.
- READ_WRITE: used for address ranges, which are initialized by the startup code at runtime. Memory area defined with this qualifier will be initialized with 0 at application startup.
- NO_INIT: used for address ranges, where read write accesses are allowed. Memory area defined with this qualifier will not be initialized with 0 at application startup. This may be useful if your target has a battery buffered RAM or to speedup application startup.
- PAGED: used for address ranges, where read write accesses are allowed. Memory area defined with this qualifier will not be initialized with 0 at application startup. Additionally, the linker will not control if there is an overlap between segments defined with the PAGED qualifier. When overlapped segments are used, it is the user's responsibility to select the correct page before accessing the data allocated on a certain page.

Qualifier Handling

Qualifier	Initialized Variables	Non-Initialized Variables	Constants	Code
READ_ONLY	not applicable (1)	not applicable (1)	content written to target address	content written to target address
READ_WRITE	content written into copy down area, together with info where to copy it at startup. Area contained in zero out information (3) (4)	Area contained in zero out information (4)	content written into copy down area, together with info where to copy it at startup. Area contained in zero out information (3) (4)	not applicable (1) (2)

Qualifier	Initialized Variables	Non-Initialized Variables	Constants	Code
NO_INIT	not applicable (1)	just handled as allocated. Nothing generated.	not applicable (1)	not applicable (1)
PAGED	not applicable (1)	just handled as allocated. Nothing generated.	not applicable (1)	not applicable (1)

1. These cases are not intended. The linker does however allow some of them. If so, the qualifier controls what is written into the application.
2. To allocate code in a RAM area, for example for testing purposes, declare this area as READ_ONLY.
3. Initialized objects and constant in READ_WRITE sections do need additionally to the RAM memory, also space in the copy down area. The copy down contains the information how the object is initialized in the startup code.
4. The zero out information consist of the information which areas should be initialized with 0 at startup. Because the zero out contains only an address and a size per area, it is usually much smaller than a copy down area, which also contains the (non zero) content of the objects to be initialized.

Example

```
SEGMENTS
  ROM   = READ_ONLY  0x1000 SIZE 0x2000;
  CLOCK = NO_INIT    0xFF00 TO   0xFFFF;
  RAM   = READ_WRITE 0x3000 TO   0x3EFF;
  Page0 = PAGED      0x4000 TO   0x4FFF;
  Page1 = PAGED      0x4000 TO   0x4FFF;
END
```

In the previous example:

- the segment ’ROM’ is a READ_ONLY memory area. It starts at address 0x1000 and its size is 0x2000 bytes (from address 0x1000 to 0x2FFF).

- The segment ’RAM’ is a READ_WRITE memory area. It starts at address 0x3000 and ends at 0x3FFF (size = 0x1000 bytes). All variables allocated in this segment will be initialized with 0 at application startup.
- The segment ’CLOCK’ is a READ_WRITE memory area. It starts at address 0xFF00 and ends at 0xFFFF (size = 100 bytes). Variables allocated in this segment will not be initialized at application startup.
- The segments ’Page0’ and ‘Page1’ is a READ_WRITE memory area. These are overlapping segments. It is the user responsibility to select the correct page before accessing any data allocated in one of these segment. Variables allocated in this segment will not be initialized at application startup.

Defining an Alignment Rule

An alignment rule can be associated with each segment in the application. This may be useful when specific alignment rules are expected on a certain memory range, because of hardware restriction for example.

An Alignment rule can be specified as follows:

```
ALIGN [<defaultAlignment>] [{‘[‘(<Number>|
    <Number> ‘TO’ <Number>|
    (‘<‘ | ’>’ | ’<=’ | ’>=’)<Number>)’]:’<alignment>}]
```

defaultAlignment: is used to specify the alignment factor for objects, which do not match any condition in the following alignment list. If there is no alignment list specified, the default alignment factor applies to all objects allocated in the segment. The default alignment factor is optional.

Example

```
SEGMENTS
    RAM_1  = READ_WRITE 0x800 TO 0x8FF
             ALIGN 2 [1:1];
    RAM_2  = READ_WRITE 0x900 TO 0x9FF
             ALIGN [2 TO 3:2] [>= 4:4];
    RAM_3  = READ_WRITE 0xA00 TO 0xAFF
             ALIGN 1 [>=2:2];
END
```

In the previous example:

- Inside of segment RAM_1, all objects which size is equal to 1 byte are aligned on 1 byte boundary and all other objects are aligned on 2 bytes boundary.

- Inside of segment RAM_2, all objects which size is equal to 2 or 3 bytes are aligned on 2 bytes boundary and all objects which size is bigger or equal to 4 are aligned on 4 bytes boundary. Objects which size is 1 byte follow the default processor alignment rule.
- Inside of segment RAM_3, all objects which size is equal bigger or equal to 2 bytes are aligned on 2 bytes boundary and all other objects are aligned on 1 bytes boundary.

Alignment rules applying during object allocation are described in chapter alignment.

Defining a Fill Pattern

An fill pattern can be associated with each segment in the application. This may be useful to automatically initialize not initialized variables in the segments with a predefined pattern.

An Fill pattern can be specified as follows:

```
FILL <HexByte> {<HexByte>}
```

Example

```
SEGMENTS
    RAM_1  = READ_WRITE 0x800 TO 0x8FF
             FILL 0xAA 0x55;
END
```

In the previous example, non-initialized objects and filling bytes are initialized with the pattern 0xAA55.

If the size of an object to initialize is higher than the size of the specified pattern, the pattern is repeated as many time as required to fill the objects. In the previous example, an object which size is 4 bytes will be initialized with 0xAA55AA55.

If the size of an object to initialize is smaller than the size of the specified pattern, the pattern is truncated to match exactly the size of the object. In the previous example, an object which size is 1 byte will be initialized with 0xAA.

When the value specified in an element of a fill pattern does not fit in a byte, it is truncated to a byte value.

Example

```
SEGMENTS
```

```
    RAM_1   = READ_WRITE 0x800 TO 0x8FF
              FILL 0xAA55;
END
```

In the previous example, non-initialized objects and filling bytes are initialized with the pattern 0x55. The specified fill pattern is truncated to a 1-byte value.

Fill patterns are useful to assign an initial value to the padding bytes inserted between two objects during object allocation. This allows marking from the unused position with a specific marker and detecting them inside of the application.

For example, unused position inside of a code section can be initialized with the hexadecimal code for the NOP instruction.

Optimizing Constants with Common Code

Constants having the same byte pattern can be allocated to the same addresses. The most common usage is to allocate some string in another string.

Example

```
const char* hwstr="Hello World";
const char* wstr= "World";
```

The string “World” is exactly contained in the string “Hello World”. When the constants are optimized, wstr will point to hwstr+6.

In the HIWARE format, the linker does only optimize strings. In the ELF format, however all constant objects including strings, constants and code can be optimized.

For all segments it can be specified if code or data (only constants and strings) should be optimized. If nothing is specified, the default is controlled with the option -Cocc.

Example

C-Source File

```
void print1(void) {
  printf("Hello");
}
void print2(void) {
  printf("Hello");
}
```

Prm File:

```
SECTIONS
    ...
    MY_ROM = READ_ONLY 0x9000 TO 0xFEFF DO_OVERLAP_CONSTS CODE DATA;
END
```

Because data is optimized, the string “Hello” will only be once in the ROM-image. Because code and data is optimized, also the function print1 and print2 are allocated at the same address. Note however, if only code should be optimized (this in not the case here), then print1 and print2 would not be optimized because they were using a different instance of the string “Hello”.

If code is optimized the linker issues the warning “L1951: Function print1 is allocated inside of print2 with offset 0. Debugging may be affected”. This warning is issued because the debugger cannot distinguish between print1 and print2. So the wrong function might be displayed while debugging. This does however not affect the runtime behavior.

The linker does detect certain branch distance optimizations done by the compiler because of the special fixups used. If the linker detects such a case, both the caller and the callee are not moved into other functions. However, other functions can still be moved into them. Also switching off this compiler optimizations can produce smaller applications, if the compiler optimizations does prevent linker optimizations.

One important case of this optimization are C++ applications. In C++ several language constructs result in identical functions in different compilation units. Different instances of the same template might have identical code. Compiler generated functions and inline functions, which were not actually inlined are defined in every compilation unit. Finally, constants defined in header files are static in C++. So they are also contained in every object file once.

STACKSIZE

STACKSIZE: Define Stack Size

Syntax

```
STACKSIZE Number
```

Description

The STACKSIZE command is optional in a prm file and it cannot be specified several times. Additionally, you cannot specify both STACKTOP and STACKSIZE command in a prm file.

The STACKSIZE command defines the size requested for the stack. We recommend using this command if you do not care where the stack is allocated but only how large it is.

When the stack is defined trough a STACKSIZE command alone, the stack is placed next to the section .data.

Note: In the HIWARE object file format, the synonym STACK instead of STACKSIZE is allowed too. This is for compatibility only, and may be removed in a future version.

Example

```
SECTIONS
  MY_RAM = READ_WRITE 0xA00 TO 0xAFF;
  MY_ROM = READ_ONLY  0x800 TO 0x9FF;
PLACEMENT
  DEFAULT_ROM INTO MY_ROM;
  DEFAULT_RAM INTO MY_RAM;
END
STACKSIZE 0x60
```

In the previous example, if the section .data is 4 bytes wide (from address 0xA00 to 0xA03), the section .stack is allocated next to it, from address 0xA63 down to address 0xA04. The stack initial value is set to 0xA62.

When the stack is defined trough a STACKSIZE command associated with the placement of the .stack section, the stack is supposed to start at the segment start address incremented by the specified value and is defined down to the start address of the segment, where .stack has been placed.

Example

```
SECTIONS
  MY_STK = NO_INIT    0xB00 TO 0xBFF;
  MY_RAM = READ_WRITE 0xA00 TO 0xAFF;
  MY_ROM = READ_ONLY  0x800 TO 0x9FF;
PLACEMENT
```

```
  DEFAULT_ROM INTO MY_ROM;
  DEFAULT_RAM INTO MY_RAM;
  SSTACK      INTO MY_STK;
END
STACKSIZE 0x60
```

In the previous example, the section SSTACK is allocated from address 0xB5F down to address 0xB00. The stack initial value is set to 0xB5E.

STACKTOP

STACKTOP: Define Stack Pointer Initial Value

Syntax

```
STACKTOP Number
```

Description

The STACKTOP command is optional in a prm file and it cannot be specified several times. Additionally, you cannot specify both STACKTOP and STACKSIZE command in a prm file.

The STACKTOP command defines the initial value for the stack pointer

Example

If STACKTOP is defined as

```
STACKTOP 0xBFF
```

the stack pointer will be initialized with 0xBFF at application startup.

When the stack is defined trough a STACKTOP command alone, a default size is affected to stack. This size depends on the processor and is big enough to store the target processor PC.

When the stack is defined trough a STACKTOP command associated with the placement of the .stack section, the stack is supposed to start at the specified address, and is defined down to the start address of the segment, where .stack has been placed.

Example

```
SEGMENTS
  MY_STK = NO_INIT    0xB00 TO 0xBFF;
  MY_RAM = READ_WRITE 0xA00 TO 0xAFF;
  MY_ROM = READ_ONLY  0x800 TO 0x9FF;
END
PLACEMENT
  DEFAULT_ROM   INTO MY_ROM;
  DEFAULT_RAM   INTO MY_RAM;
  SSTACK        INTO MY_STK;
END
STACKTOP 0xB7E
```

In the previous example, the stack pointer will be defined from address 0xB7E down to address 0xB00.

START

START: Specify the ROM Start (HIWARE)

Syntax

```
START  Address
```

Description

This is a command supported in ‘old-style’ linker parameter files and will be not supported in a future release.

With this command the default ROM begin can be specified. The specified address has to be in hexadecimal notation. Internally this command is translated into:

```
START 0x????' => 'DEFAULT_ROM INTO READ_ONLY 0x???? TO 0x????
```

Note that because the end address is of DEFAULT_ROM is not known, the linker tries to specify/find out the end address itself. Because this is not a very transparent behavior, this command will not be supported any more.

If you get an error message during linking that START is not defined: The reason could be that there is no application entry point visible for the linker, e.g. the ‘main’ routine is defined as static.

Example

```
START 0x1000
```

VECTOR

VECTOR: Initialize Vector Table

Syntax

```
VECTOR  (InitByAddr | InitByNumber)
```

Description

The VECTOR command is optional in a prm file and it can be specified several times.

A vector is a small piece of memory, having the size of a function address. This command allows the user to initialize the processor’s vectors while downloading the absolute file.

A VECTOR command consist in a vector location part (containing the location of the vector) and a vector target part (containing the value to store in the vector).

The vector location part can be specified:

- through a vector number. The mapping of vector numbers to addresses is target specific.
  - For targets with vectors starting at 0, the vector is allocated at <Number> * <Size of a Function Pointer>.
  - For targets with vectors located from 0xFFFE and allocated downwards, VECTOR 0 maps to 0xFFFE. In general the address is 0xFFFE- <Number> * 2.
  - For HC05 and St7 the environment variable RESETVECTOR specifies the address of VECTOR 0. All other vectors are calculated depending on it. As default, address 0xFFFE is used.

  – For all other supported targets, VECTOR numbers do automatically map to vector locations natural for this target.
- through a vector address. In this case the keyword ADDRESS must be specified in the vector command.

The vector target part can be specified:

- as a function name
- as an absolute address.

Example

```
VECTOR ADDRESS 0xFFFE _Startup
VECTOR ADDRESS 0xFFFC 0xA00
VECTOR 0 _Startup
VECTOR 1 0xA00
```

In the previous example, if the size of a function pointer is coded on two bytes:

- The vector located at address 0xFFFE is initialized with the address of the function ‘_Startup’.
- The vector located at address 0xFFFC is initialized with the absolute address 0xA00.
- The address of vector numbers is target specific.
  For a HC16, vector number 0 (located at address 0x000) is initialized with the address of the function ‘_Startup’.
  For a HC08 or HC12 vector number 0 is located at address 0xFFFE.
- The address of vector numbers is target specific.
  For a HC16, the vector number 1 (located at address 0x002) is initialized with the absolute address 0xA00.
  For a HC08 or HC12 vector number 1 is located at address 0xFFFC.

You can specify an additional offset when the vector target is a function name. In this case the vector will be initialized with the address of the object + the specified offset.

Example

```
VECTOR ADDRESS 0xFFFE CommonISR OFFSET 0x10
```

In the previous example, the vector located at address 0xFFFE is initialized with the address of the function ‘CommonISR’ + 0x10 Byte. If ‘CommonISR’ starts at address 0x800, the vector will be initialized with 0x810.

This notation is very useful for common interrupt handler.

All objects specified in a VECTOR command are entry points in the application. They are always linked with the application, as well as the objects they refer to.

9

Sections (ELF)

The section concept gives the user complete control over allocation of objects in memory. A section is a named group of global objects (variables or functions) associated with a certain memory area that may be non-contiguous. The objects belonging to a section are allocated in its associated memory range. This chapter describes the use of segmentation in detail.

There are many different ways to make use of the section concept, the most important being

- Distribution of two or more groups of functions and other read-only objects to different ROMs.
- Allocation of a single function or variable to a fixed absolute address (for example, to access processor ports using high level language variables).
- Allocation of variables in memory locations where special addressing modes may be used.

Terms: Segments and Sections

A *Section* is a named group of global objects declared in the source file, that is, functions and global variables.

A *Segment* is a not necessarily contiguous memory range.

In the linker's parameter file, each section is associated with a segment so the linker knows where to allocate the objects belonging to a section.

Definition of Section

A section definition always consists of two parts: the definition of the objects belonging to it, and the memory area(s) associated with it, called segments. The first is necessarily done in the source files of the application using pragmas or directive, please see *Compiler or Assembler* Manual. The second is done in the parameter file

using the SEGMENTS and PLACEMENT commands (Please see section *The Syntax of the Parameter File*).

Some predefined sections are handled in a particular way.

Predefined Sections

There are a couple of predefined section names which can be grouped into sections named by the runtime routines

- Sections for other things than variables and functions: .rodata1, .copy, .stack.
- Sections for grouping large sets of objects: .data, .text.
- A section for placing objects initialized by the linker: .startData.
- A Section to allocate read-only variables: .rodata

NOTE	The sections .data and .text provide default sections for allocating objects.

Subsequently we will discuss each of these predefined sections.

.rodata1 All string literals (for example, *This is a string*) are allocated in section .rodata1. If this section is associated with a segment qualified as READ_WRITE, the strings are copied from ROM to RAM at startup.

.rodata Any constant variable (declared as const in a C module or as DC in an assembler module), which is not allocated in a user-defined section, is allocated in section .rodata. Usually, the .rodata section is associated with READ_ONLY segment.

If this section is not mentioned in the PLACEMENT block in the parameter file, these variables are allocated next to the section .text.

.copy Initialization data belongs to section .copy. If a source file contains the declaration

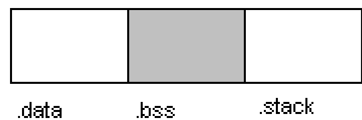
```
int a[] = {1, 2, 3};
```

the hex string 000100020003 (6 bytes), which is copied to a location in RAM at program startup, belongs to segment .copy.

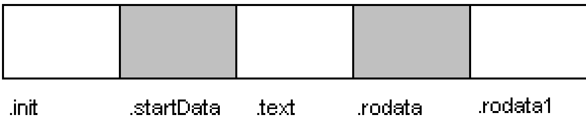
If the `.rodata1` section is allocated to a `READ_WRITE` segment, all strings also belong to the `.copy` section. Any objects in this section are copied at startup from ROM to RAM.

`.stack` The runtime stack has its own segment named `.stack`. It should always be allocated to a `READ_WRITE` segment.

`.data` This is the default section for all objects normally allocated to RAM. It is used for variables not belonging to any section or to a section not assigned a segment in the `PLACEMENT` block in the linker's parameter file. If any of the sections `.bss` or `.stack` is not associated with a segment, these sections are included in the `.data` memory area in the following order:



`.text` This is the default section for all functions. If a function is not assigned to a certain section in the source code or if its section is not associated with a segment in the parameter file, it is automatically added to section `.text`.I f any of the sections `.rodata, .rodata1, .startData` or `.init` is not associated with a segment, these sections are included in the `.text` memory area in the following order:



`.startData` The startup description data initialized by the linker and used by the startup routine is allocated to segment `.startData`. This section must be allocated to a `READ_ONLY` segment.

`.init` The application entry point is stored in the section `.init`. This section also has got to be associated with a `READ_ONLY` segment.

`.overlap` Compilers using pseudo-statically variables for locals are allocating these variables in .overlap. Variables of functions not depending on each other may be allocated at the same place. For details see the chapter Overlapping Locals. This section must be associated with a `NO_INIT` segment.

NOTE	The .data and .text sections must always be associated with a segment.

10

Segments (HIWARE)

The segment concept gives the user complete control over allocation of objects in memory. A segment is a named group of global objects (variables or functions) associated with a certain memory area that may be non-contiguous. The objects belonging to a segment are allocated in its associated memory range. This chapter describes the use of segmentation in detail.

There are many different ways to make use of the segment concept, the most important being

- Distribution of two or more groups of functions and other read-only objects to different ROMs.
- Allocation of a single function or variable to a fixed absolute address (for example, to access processor ports using high level language variables).
- Allocation of variables in memory locations where special addressing modes may be used.

Terms: Segments and Sections (HIWARE)

A *Segment* is a named group of global objects declared in the source file, i.e. functions and global variables.

A *Section* is a not necessarily contiguous memory range.

In the linker's parameter file, each segment is associated with a section so the linker knows where to allocate the objects belonging to a segment.

Definition of Segment (HIWARE)

A segment definition always consists of two parts: the definition of the objects belonging to it, and the memory area(s) associated with it, called sections. The first is necessarily done in the source files of the application using pragmas or directive, please see *Compiler or Assembler* Manual. The second is done in the parameter file

using the SECTIONS and PLACEMENT commands (Please see section *The Syntax of the Parameter File*).

Some predefined sections are handled in a particular way.

Predefined Segments

There are a couple of predefined section names which can be grouped into sections named by the runtime routines

- Segments for other things than variables and functions: STRINGS, COPY, SSTACK.
- Segments for grouping large sets of objects: DEFAULT_RAM, DEFAULT_ROM.
- A Segment for placing objects initialized by the linker: STARTUP.
- A Segment to allocate read-only variables: ROM_VAR

NOTE	The Segments DEFAULT_RAM and DEFAULT_ROM provide default segments for allocating objects.

Subsequently we will discuss each of these predefined segments.

STRINGS All string literals (e.g. “This is a string”) are allocated in segment STRINGS. If this segment is associated with a segment qualified as READ_WRITE, the strings are copied from ROM to RAM at startup.

ROM_VAR Any constant variable (declared as const in a C module or as DC in an assembler module), which is not allocated in a user-defined segment, is allocated in segment ROM_VAR. Usually, the ROM_VAR segment is associated with READ_ONLY section.

If this segment is not mentioned in the PLACEMENT block in the parameter file, these variables are allocated next to the segment DEFAULT_ROM.

FUNCS Any function code, which is not allocated in a user-defined segment, is allocated in segment FUNCS. Usually, the FUNCS segment is associated with READ_ONLY section.

COPY Initialization data belongs to segment COPY. If a source file contains the declaration

```
int a[] = {1, 2, 3};
```

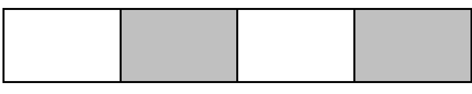
the hex string 000100020003 (6 bytes), which is copied to a location in RAM at program startup, belongs to segment COPY.

If the STRINGS segment is allocated to a READ_WRITE section, all strings also belong to the COPY segment. Any objects in this segment are copied at startup from ROM to RAM.

SSTACK The runtime stack has its own segment named SSTACK. It should always be allocated to a READ_WRITE section.

DEFAULT_RAM This is the default segment for all objects normally allocated to RAM. It is used for variables not belonging to any segment or to a segment not assigned a section in the PLACEMENT block in the linker's parameter file. If the segment SSTACK is not associated with a section, it is appended to the DEFAULT_RAM memory area.

DEFAULT_ROM This is the default segment for all functions. If a function is not assigned to a certain segment in the source code or if its segment is not associated with a section in the parameter file, it is automatically added to segment DEAFULT_ROM. If any of the segments _PRESTART, STARTUP or COPY is not associated with a section, these segments are included in the DEFAULT_ROM memory area in the following order:



STARTUP The startup description data initialized by the linker and used by the startup routine is allocated to segment STARTUP. This segment must be allocated to a READ_ONLY section.

_PRESTART The application entry point is stored in the segment _PRESTART. This segment also has got to be associated with a READ_ONLY section.

_OVERLAP This segment contains local variables, which are by the compiler pseudo-statically for non-reentrant functions.
The linker analyzes the call graph (that is, it keeps track of which function calls which other functions) and chooses memory areas in the _OVERLAP segment that are distinct if it detects a call dependency between two functions. If it doesn't detect such a dependency, it may overlap the memory areas used for two separate functions' local variables (hence the name of the segment).

There are cases in which the linker cannot exactly determine whether a function calls some other function, especially in the presence of function pointers. If the linker detects a conflict between two functions, it issues an error message.
In the ELF object file format, the name .overlap is a synonym for _OVERLAP.
For details of the usage of this segment, see also chapter Overlapping Locals.

NOTE The DEFAULT_RAM and DEFAULT_ROM segments must always be associated with a section.

11

Examples

Examples 1 and 2 illustrate the use of sections to control allocation of variables and functions precisely.

Example 1

Distributing code into two different ROMs:

```
LINK first.ABS
NAMES first.o strings.o startup.o END
STACKSIZE 0x200
SECTIONS
  ROM1 = READ_ONLY 0x4000 TO 0x4FFF;
  ROM2 = READ_ONLY 0x8000 TO 0x8FFF;
PLACEMENT
  DEFAULT_ROM INTO ROM1, ROM2;
  DEFAULT_RAM INTO READ_WRITE 0x1000 TO 0x1FFF;
END
```

Example 2

Allocation in battery buffered RAM:

```
/* Extract from source file "bufram.c" */
#pragma DATA_SEG Buffered_RAM
  int done;
  int status[100];
#pragma DATA_SEG DEFAULT
/* End of extract from "bufram.c" */
```

SmartLinker parameter file:

```
LINK bufram.ABS
NAMES
```

```
  bufram.o startup.o
END
STACKSIZE 0x200
SECTIONS
  BatteryRAM = NO_INIT    0x1000 TO 0x13FF;
  MyRAM      = READ_WRITE 0x5000 TO 0x5FFF;
PLACEMENT
  DEFAULT_ROM  INTO READ_ONLY 0x2000 TO 0x2800;
  DEFAULT_RAM  INTO MyRAM;
  Buffered_RAM INTO BatteryRAM;
END
```

12

Program Startup

NOTE This section deals with advanced material. First time users of HI-CROSS+ may skip this section; standard startup modules taking care of the common cases are delivered together with the HI-CROSS+ programs and examples. It suffices to include the startup module in the files to link in the parameter file. For more information about the names of the startup modules and the different variants see the file Startup.txt in directory LIB.

Prior to calling the application's root function (`main`), one must:

- initialize the processor's registers,
- zero out memory and
- copy initialization data from ROM to RAM.

Depending on the processor and the application's needs different startup routines may be necessary.

In HI-CROSS+, there are standard startup routines for every processor and memory model. They're easy to adapt to your particular needs because all these startup routines are based on a startup descriptor containing all information needed. Different startup routines only differ in the way they make use of that information.

The Startup Descriptor (ELF)

The startup descriptor of the linker is declared as

```
typedef struct{
  unsigned char *far beg;int size;
} _Range;

typedef struct{
  int size; unsigned char * far dest;
```

```
} _Copy;

typedef void (*_PFunc)(void);

typedef struct{
  _PFunc  *startup;   /* address of startup desc */
} _LibInit;

typedef struct _Cpp {
  _PFunc  initFunc;      /* address of init function */
} _Cpp;


extern struct _tagStartup {
   unsigned char   flags;
   _PFunc          main;
   unsigned short  stackOffset;
   unsigned short  nofZeroOuts;
   _Range          *pZeroOut;
   _Copy           *toCopyDownBeg;
   unsigned short  nofLibInits;
   _LibInit        *libInits;
   unsigned short  nofInitBodies;
   _Cpp            *initBodies;
   unsigned short  nofFiniBodies;
   _Cpp            *finiBodies;
} _startupData;
```

The linker expects somewhere in your application a declaration of the variable `_startupData`, that is:

```
struct _tagStartup _startupData;
```

The fields of this `struct` are initialized by the linker and the `_startupData` is allocated in ROM in section `.startData`. If there is no declaration of this variable, the linker does not create a startup descriptor. In this case, there is no `.copy` section, and the stack is not initialized. Furthermore, global C++ constructor and ROM libraries are not initialized.

The fields have the following semantics:

`flags` Contains some flags, which may be used to detect special condition at startup. Currently two bits are used, as shown in Table 12.1:

Table 12.1 Bits description

Bit #	Set if...
0	The application has been linked as a ROM library
1	There is no stack specification

The bit 1 (with mask 2) is tested in the startup code, to detect if the stack pointer should be initialized.

main is a function pointer set to the application's root function. In a C program, this usually is function `main` unless there is a `MAIN` entry in the parameter file specifying some other function as being the root. In a ROM library, this field is zero. The standard startup code jumps to this address once the whole initialization is over.

stackOffset is valid only if `(flags & 2) == 0`. This field contains the initial value of the stack pointer.

nofZeroOuts is the number of `READ_WRITE` segments to fill with zero bytes at startup.

This field is not required if you do not have any RAM memory area, which should be initialized at startup. Be careful, when this field is not present in the startup structure, the field **pZeroOut** must not be present either.

pZeroOut is a pointer to a vector with elements of type `_Range`. It has exactly **nofZeroOuts** elements, each describing a memory area to be cleared. This field is not required if you do not have any RAM memory area, which should be initialized at startup. Be careful, when this field is not present in the startup structure, the field **nofZeroOuts** must not be present either.

toCopyDownBeg contains the address of the first item which must be copied from ROM to RAM at runtime. All data to be copied is stored in a contiguous piece of ROM memory and has the following format:

```
CopyData = {Size[t] TargetAddr {Byte}^Size Alignment} 0x0[t].
Alignment= 0x0[0..7].
```

The size is a binary number whose most significant byte is stored first. This field is not required if you do not have any RAM memory area, which should be initialized at startup. The alignment is used to align the next size and TargetAddr field. The number of alignment bytes depends on the processors capability to access non aligned data. For small processors, there is usually no alignment. The size t of $\texttt{Size}_{[t]}$ and $\texttt{0x0}_{[t]}$ does depend on the target processor and memory model.

nofLibInits is the number of ROM libraries linked with the application that must be initialized at startup. This field is not required if you do not link any ROM library with your application. Be careful, when this field is not present in the startup structure, the field **libInits** must not be present.

libInits is a vector of pointers to the `_startupData` records of all ROM libraries in the application. It has exactly **nofLibInits** elements. These addresses are needed to initialize the ROM libraries. This field is not required if you do not link any ROM library with your application. Be careful, when this field is not present in the startup structure, the field **nofLibInits** must not be present

nofInitBodies is the number of C++ global constructors, which must be executed prior to invoking the application root function. This field is not required if your application does not contain any C++ module. Be careful, when this field is not present in the startup structure, the field **initBodies** must not be present

initBodies is a pointer to a vector of function pointers containing the addresses of the global C++ constructors in the application, sorted in the order they have to be called. It has exactly **nofInitBodies** elements. If an application does not contain any C++ modules, the vector is empty. This field is not required if your application does not contain any C++ module. Be careful, when this field is not present in the startup structure, the field **nofInitBodies** must not be present either.

nofFiniBodies is the number of C++ global destructors, which must be executed after the invocation of the application root function. This field is not required if your application does not contain any C++ module. Be careful, when this field is not present in the startup structure, the field **finiBodies** must not be present either. If the application root function does not return, **nofFiniBodies** and **finiBodies** can both be omitted.

finiBodies is a pointer to a vector of function pointers containing the addresses of the global C++ destructors in the application, sorted in the order they have to be called. It has exactly **nofFiniBodies** elements. If an application does not contain any C++ modules, the vector is empty. This field is not required if your application does not contain any C++ module. Be careful, when this field is not present in the startup structure, the field **nofFiniBodies** must not be present either. If the application root function does not return, **nofFiniBodies** and **finiBodies** can both be omitted.

User Defined Startup Structure: (ELF)

The user can define his own startup structure. That means it can remove the fields, which are not required for his application, or move the fields inside of the structure. If

the user changes the startup structure, it is his responsibility to adapt the startup function to match the modification he performs.

Example

If the user does not have any RAM area to initialize at startup, no ROM libraries and no C++ modules in the application, he can define the startup structure as follows:

```
extern struct _tagStartup {
    unsigned short  flags;
    _PFunc          main;
    unsigned short  stackOffset;
} _startupData;
```

In that case the startup code must be adapted accordingly in the following way:

```
extern void near _Startup(void) {
/*  purpose:  1)  initialize the stack
              2)  call main;
    parameters: NONE */
  do { /* forever: initialize the program; call the root-procedure */
    asm{
      ; adapted for the HC12. Please modify it for other CPUS.
      LDD  _startupData.flags
      BNE  Initialize
      LDS  _startupData.stackOffset
    Initialize:
    }
    /* Here user defined code could be inserted,
       the stack can be used
    */
    /* call main() */
    (*_startupData.main)();
  } while(1); /* end loop forever */
}
```

NOTE The name of the fields in the startup structure should not be changed. The user is free to remove fields inside of the structure, but he should respect the names of the different fields, otherwise the SmartLinker will not be able to initialize the structure correctly.

User Defined Startup Routines (ELF)

There are two ways to replace the standard startup routine by one of your own:

You may provide a startup module containing a function named `_Startup` and link it with the application in place of the startup module delivered.

You can implement a function with another name as _Startup and define it as entry point for your application using the command INIT

```
INIT function_name
```

In the latter case, function `function_name` is the startup routine.

The Startup Descriptor (HIWARE)

The startup descriptor of the linker is declared as

```
typedef struct{
  unsigned char  *beg; int size;
} _Range;

typedef void (*_PFunc)(void);

extern struct _tagStartup{
  unsigned           flags;
  _PFunc             main;
  unsigned           dataPage;
  long               stackOffset;
  int                nofZeroOuts;
  _Range             *pZeroOut;
  long               toCopyDownBeg;
  _PFunc             *mInits;
  struct _tagStartup *libInits;
} _startupData;
```

The linker expects somewhere in your application a declaration of the variable `_startupData`, that is:

```
struct _tagStartup _startupData;
```

The fields of this `struct` are initialized by the linker and the `struct` is allocated in ROM in segment `STARTUP`. If there is no declaration of this variable, the linker does not create a startup descriptor. In this case, there is no `COPY` segment, and the

stack is not initialized. Furthermore, global C++ constructor and ROM libraries are not initialized.

The fields have the following semantics:

flags Contains some flags, which may be used to detect special condition at startup. Currently two bits are used, as shown in Table 12.2:

Table 12.2 Bits description

Bit #	Set if...
0	The application has been linked as a ROM library
1	There is no stack specification

This flag is tested in the startup code, to detect if the stack pointer should be initialized.

main is a function pointer set to the application's root function. In a C program, this usually is function `main` unless there is a `MAIN` entry in the parameter file specifying some other function as being the root. In a ROM library, this field is zeroed out. The standard startup code jumps to this address once the whole initialization is over.

datapage is only used for processor having paged memory and memory models supporting only one page. In this case, dataPage gives the page.

stackOffset is valid only if `flags == 0`. This field contains the initial value of the stack pointer.

nofZeroOuts is the number of `READ_WRITE` segments to fill with zero bytes at startup.

pZeroOut is a pointer to a vector with elements of type `_Range`. It has exactly **nofZeroOuts** elements, each describing a memory area to be cleared.

toCopyDownBeg contains the address of the first item which must be copied from ROM to RAM at runtime. All data to be copied is stored in a contiguous piece of ROM memory and has the following format:

$\texttt{CopyData = \{Size}_{[2]}\ \texttt{TargetAddr \{Byte\}}^{\texttt{Size}}\texttt{\} 0x0}_{[2]}.$

The size is a binary number whose most significant byte is stored first.

libInits is a pointer to an array of pointers to the _startupData records of all ROM libraries in the application. These addresses are needed to initialize the ROM libraries. To specify the end of the array, the last array element contains the value 0x0000ffff.

mInits is a pointer to an array of function pointers containing the addresses of the global C++ constructors in the application, sorted in the order they have to be called. The array is terminated by a single zero entry.

User Defined Startup Routines (HIWARE)

There are two ways to replace the standard startup routine by one of your own:

You may provide a startup module containing a function named `_Startup` and link it with the application in place of the startup module delivered.

You can implement a function with another name as _Startup and define it as entry point for your application using the command INIT

```
INIT function_name
```

In the latter case, function `function_name` is the startup routine.

Example of Startup Code in ANSI-C

Normally the startup code delivered with the compiler is provided in HLI for code efficiency reasons. But there is also a version in ANSI-C available in the library directory (startup.c and startup.h). You may use this startup for your own modifications or just to get familiar with the startup concept. The code printed here may vary depending on the actual implementation.

Header File startup.h:

```
/***************************************************
  FILE    : startup.h
  PURPOSE : data structures for startup
  LANGUAGE: ANSI-C
****************************************************/
#ifndef STARTUP_H
#define STARTUP_H
#ifdef __cplusplus
extern "C" {
#endif
/*
  the following data structures contain the data needed to
  initialize the processor and memory
*/

typedef struct{
```

```
  unsigned char *beg;
  int size;      /* [beg..beg+size] */
} _Range;

typedef struct _Copy{
  int size;
  unsigned char * dest;
} _Copy;

typedef struct _Cpp {
  _PFunc  initFunc;      /* address of init function */
} _Cpp;

typedef void (*_PFunc)(void);
typedef struct _LibInit{
  struct _tagStartup  *startup; /* address of startup desc */
} _LibInit;
#define STARTUP_FLAGS_NONE    0
#define STARTUP_FLAGS_ROM_LIB (1<<0) /* ROM library */
#define STARTUP_FLAGS_NOT_INIT_SP (1<<1) /* init stack */
#ifdef __ELF_OBJECT_FILE_FORMAT__
/* ELF/DWARF object file format */
/* attention: the linker scans for this structs */
/* to obtain the available fields and their sizes. */
/* So do not change the names in this file. */

extern struct _tagStartup {
  unsigned char flags;         /* STARTUP_FLAGS_xxx     */
  _PFunc         main;         /* first user fct        */
  unsigned short stackOffset;  /* initial stack pointer */
  unsigned short nofZeroOuts;  /* number of zero outs   */
  _Range         *pZeroOut;    /* vector of zero outs   */
  _Copy          *toCopyDownBeg; /* copy down start     */
  unsigned short nofLibInits;    /* number of ROM Libs  */
  _LibInit       *libInits;      /* vector of ROM Libs  */
  unsigned short  nofInitBodies; /* number of C++ inits */
  _Cpp           *initBodies;    /* C+ init funcs       */
  unsigned short  nofFiniBodies; /* number of C++ dtors */
  _Cpp           *finiBodies;    /* C+ dtors funcs      */
} _startupData;

#else /* HIWARE format */

extern struct _tagStartup {
  unsigned  flags;        /* STARTUP_FLAGS_xxx */
```

```
  _PFunc    main;          /* starting point of user code */
  unsigned  dataPage;      /* page where data begins */
  long      stackOffset;   /* initial stack pointer */
  int       nofZeroOuts;   /* number of zero out ranges */
  _Range    *pZeroOut;     /* prt to zero out descriptor */
  long      toCopyDownBeg;/* address of copydown descr */
  _PFunc    *mInits;       /* ptr to C++ init fcts */
  _LibInit  *libInits;     /* ptr to ROM Lib descriptors */
} _startupData;

#endif

extern void _Startup(void);    /* execution begins here */
/*-----------------------------------------------*/
#ifdef __cplusplus
 }
#endif
#endif /* STARTUP_H */
```

Implementation File startup.c

```
/*************************************************
  FILE        : startup.c
  PURPOSE     : standard startup code
  LANGUAGE    : ANSI-C / HLI
 *************************************************/
#include <hidef.h>
#include <startup.h>
/*************************************************/
struct _tagStartup _startupData; /* startup info   */
/*-----------------------------------------------*/
static void ZeroOut(struct _tagStartup *_startupData) {
/* purpose: zero out RAM-areas where data is allocated.*/
  int i, j;
  unsigned char *dst;
  _Range *r;
  r = _startupData->pZeroOut;
  for (i=0; i<_startupData->nofZeroOuts; i++) {
    dst = r->beg;
    j = r->size;
    do {
      *dst = '\0'; /* zero out */
      dst++;
      j--;
    } while(j>0);
```

```
    r++;
  }
}
/*------------------------------------------------------*/
static void CopyDown(struct _tagStartup *_startupData) {
/* purpose: zero out RAM-areas where data is allocated.
   this initializes global variables with their values,
    e.g. 'int i = 5;' then 'i' is here initialized with '5' */
  int i;
  unsigned char *dst;
  int *p;
  /* _startupData.toCopyDownBeg  ---> */
  /*  {nof(16) dstAddr(16) {bytes(8)}^nof} Zero(16) */
  p = (int*)_startupData->toCopyDownBeg;
  while (*p != 0) {
    i = *p; /* nof */
    p++;
    dst = (unsigned char*)*p; /* dstAddr */
    p++;
    do {
      /* p points now into 'bytes' */
      *dst = *((unsigned char*)p); /* copy byte-wise */
      dst++;
      ((char*)p)++;
      i--;
    } while (i>0);
  }
}
/*-----------------------------------------------------*/
static void CallConstructors(struct _tagStartup *_startupData) {
/* purpose: C++ requires that the global constructors have
   to be called before main.
   This function is only called for C++ */
#ifdef __ELF_OBJECT_FILE_FORMAT__
  short i;
  _Cpp *fktPtr;

  fktPtr = _startupData->initBodies;
  for (i=_startupData->nofInitBodies; i>0; i--) {
    fktPtr->initFunc(); /* call constructors */
    fktPtr++;
  }
#else
  _PFunc *fktPtr;
  fktPtr = _startupData->mInits;
```

```
  if (fktPtr != NULL) {
    while(*fktPtr != NULL) {
      (**fktPtr)(); /* call constructors */
      fktPtr++;
    }
  }
#endif
}
/*-------------------------------------------------------*/
static void ProcessStartupDesc(struct _tagStartup *);
/*----------------------------------------------------*/
static void InitRomLibraries(struct _tagStartup *_sData) {
  /* purpose: ROM libraries have their own startup functions
     which have to be called. This is only necessary if ROM
     Libraries are used! */

#ifdef __ELF_OBJECT_FILE_FORMAT__
  short i;
  _LibInit *p;

  p = _sData->libInits;
  for (i=_sData->nofLibInits; i>0; i--) {
    ProcessStartupDesc(p->startup);
    p++;
  }
#else
  _LibInit *p;
  p = _sData->libInits;
  if (p != NULL) {
    do {
      ProcessStartupDesc(p->startup);
    } while ((long)p->startup != 0x0000FFFF);
  }
#endif
}
/*----------------------------------------------------*/
static void ProcessStartupDesc(struct _tagStartup *_sData) {
  ZeroOut(_sData);
  CopyDown(_sData);
#ifdef __cplusplus
  CallConstructors(_sData);
#endif
  if (_sData->flags&STARTUP_FLAGS_ROM_LIB) {
    InitRomLibraries(_sData);
  }
```

```
}
/*-----------------------------------------------------*/
#pragma NO_EXIT
#ifdef __cplusplus
  extern "C"
#endif
void _Startup (void) {
  for (;;) {
    asm {
      /* put your target specific initialization */
      /* (e.g. CHIP SELECTS) here */
    }
    if (!(_startupData.flags&STARTUP_FLAGS_NOT_INIT_SP)) {
      /* initialize the stack pointer */
      INIT_SP_FROM_STARTUP_DESC(); /* defined in hidef.h */
    }
    ProcessStartupDesc(&_startupData);
    (*_startupData.main)(); /* call main function */
  } /* end loop forever */
}
/*-----------------------------------------------------*/
```

13

The Map File

If linking succeeds, a protocol of the link process is written to a list file called map file. The name of the map file is the same as that of the .ABS file, but with extension .map. The map file is written to the directory given by environment variable TEXTPATH.

Map File Contents

The map file consists of many sections:

TARGET This section names the target processor and memory model.

FILE This section lists the names of all files from which objects were used or referenced during the link process. In most cases, these are the same names that are also listed in the linker parameter file between the keywords `NAMES` and `END`. If a file refers to a ROM library or a program, all object files used by the ROM library or the program are listed with indentation.

STARTUP This section lists the prestart code and the values used to initialize the startup descriptor `_startupData`. The startup descriptor is listed member by member with the initialization data at the right hand side of the member name.

SEGMENT ALLOCATION This section lists those segments, in which at least one object was allocated. At the right hand side of the segment name there is a pair of numbers, which gives the address range in which the objects belonging to the segment were allocated.

OBJECT ALLOCATION This section contains the names of all allocated objects and their addresses. The objects are grouped by module. If an address of an object is followed by the “@” sign, the object comes from a ROM library. In this case the absolute file contains no code for the object (if it is a function), but the object’s address was used for linking. If an address of a string object is followed by a dash “–”, the string is a suffix of some other string. As an example, if the strings "abc" and "bc" are present in the same program, the string "bc" is not allocated and its address is the address of "abc" plus one.

OBJECT DEPENDENCY This section lists for every function and variable that uses other global objects the names of these global objects.

DEPENDENCY TREE This section shows in a tree format all detected dependencies between functions. Overlapping Locals are also displayed at their defining function.

UNUSED OBJECTS This section lists all objects found in the object files that were not linked.

COPYDOWN This section lists all blocks that are copied from ROM to RAM at program startup.

STATISTICS This section delivers information like number of bytes of code in the application.

If linking fails because there are objects which were not found in any object file, no map file is written.

14

ROM Libraries

The SmartLinker supports linking to objects to which addresses were assigned in previous link sessions. Packages of already linked objects are called ROM libraries. Creation of a ROM library only slightly differs from the linkage of a normal program. ROM libraries can then be used in subsequent link sessions by including them into the list of files between `NAMES` and `END` (Please see section *The Semantics of the SmartLinker Commands*).

Examples for the use of ROM libraries are:

- If a set of related functions is used in different projects it may be convenient to burn these thoroughly tested library functions into ROM. We call such a set of objects (functions, variables and strings) at fixed addresses a ROM library.
- To shorten the time needed for downloading, one can build a ROM library with those modules that are known to be error free and that do not change. Such a ROM library has to be downloaded only once, before beginning the tests of the other modules of the application.
- The HI-CROSS+ system allows downloading a program while another program already is present in the target processor. The most prominent example is the monitor program. The linker facility described here enables an application program to use monitor functions.k

Creating a ROM Library

To create a ROM library, the keywords "`AS ROM_LIB`" must follow the LINK command in the linker parameter file. In the presence of the ENTRIES command, only the given objects (functions and variables) are included in the ROM library. Without an `ENTRIES` command, all exported objects are written to the ROM library. In both cases the ROM library will also contain all global objects used by those functions and variables.

Since a program cannot consist of a ROM library alone, a ROM library must not contain a function `main` or a `MAIN` or `INIT` command, and the commands `STACKSIZE` and `STACKTOP` are ignored.

Besides all the application modules which form a ROM library, the variable `_startupData` must also be defined in the ROM library. The HI-CROSS+ library includes a module containing only a definition of this variable.

ROM Libraries and Overlapping Locals

To allocate overlapping variables, all dependencies between functions have to be known at link time. For ROM libraries, the linker does not know the dependencies between all objects in the ROM library. Therefore locals of functions inside of the ROM library cannot overlap locals of the using modules. Instead, the ROM library must use a separate area for the .overlap/_OVERLAP segment which is not used in the main application.

See Also

Overlapping Locals

Using ROM Libraries

Suppressing Initialization

Linking to ROM libraries is done by adding the name of the ROM library to the list of files in the `NAMES` section of the linker parameter file. If the ROM library name is immediately followed by a dash "–" (no blank between the last character of the file name and the "–") the ROM library is not initialized by the startup routine.

An unlimited number of ROM libraries may be included in the list of files to link. As long as no two ROM libraries use the same object file, no problems should arise. If two ROM libraries contain identical objects (coming from the same object file) and both are linked in the same application, an error is reported because it is illegal to allocate the same object more than once.

Example Application

In this example, we want to build and use a ROM library named 'romlib.lib'. In this (simple) example ROM library contains only one object file with one function and one global variable. This is the header file of it:

```
/* rl.h */
#ifndef __RL_H__
```

```
#define __RL_H__

char RL_Count(void);
  /* returns the actual counter and increments it */

#endif
```

Below is the implementation. Note that somewhere in the ROM library we have to define an object named '_startupData' for the linker. This startup descriptor is used to initialize the ROM library (see below).

```
/* rom library (RL_) rl.c */
#include "rl.h"
#include <startup.h>

struct _tagStartup _startupData; /* for linker */

static char RL_counter; /* initialized to zero by startup */

char RL_Count(void) {
  /* returns the actual counter and increments it */
  return RL_counter++;
}
```

After compilation of 'rl.c' we can now link it and build a ROM library using following linker parameter file. The main difference between a normal application linker parameter file and a parameter file for ROM libraries is 'AS ROM_LIB' in the LINK command:

```
/* rl.prm */
LINK romLib.lib AS ROM_LIB

NAMES rl.o END

SECTIONS
    MY_RAM = READ_WRITE  0x4000 TO 0x43FF;
    MY_ROM = READ_ONLY   0x1000 TO 0x3FFF;

PLACEMENT
    DEFAULT_ROM, ROM_VAR, STRINGS  INTO  MY_ROM;
    DEFAULT_RAM                    INTO  MY_RAM;
END
```

In this example we have RAM from 0x4000 and ROM from 0x1000. Note that by default the Linker generates startup descriptors for ROM libraries too. The startup descriptors are used to zero out global variables or to initialize global variables with initialization values. Additionally C++ constructors and destructors may be called. This hole process is called ‘Module Initialization’ too.

To switch off Module Initialization’ for a single object file in the above linker parameter file, a dash (‘-’) has to be added at the end of each object file. For the above example this would be:

```
NAMES rl.o- END
```

After building the ROM library, the linker generates following map file (extract). Note that the linker also has generated a startup descriptor at address 0x1000 to initialize the ROM library.

```
******************************************************
STARTUP SECTION
------------------------------------------------------
Entry point: none
_startupData is allocated at 1000 and uses 44 Bytes

extern struct _tagStartup{
    unsigned flags                              3
    _PFunc   main                            103C ()
    unsigned dataPage                           0
    long     stackOffset                     4202
    int      nofZeroOuts                        1
    _Range   pZeroOut ->           4000         2
    long     toCopyDownBeg                   102C
    _PFunc   mInits   ->                     NONE
    void *   libInits ->                     NONE
} _startupData;

******************************************************
SEGMENT-ALLOCATION SECTION
------------------------------------------------------
Segmentname           Size Type    From       To  Name

FUNCS                   14  R      102E     1041  MY_ROM
COPY                     2  R      102C     102D  MY_ROM
STARTUP                 2C  R      1000     102B  MY_ROM
DEFAULT_RAM              2 R/W     4000     4001  MY_RAM
******************************************************
OBJECT-ALLOCATION SECTION
------------------------------------------------------
```

```
Type:     Name:                       Address:  Size:

MODULE:            -- rl.o --
- PROCEDURES:
          RL_Count                        102E     14

- VARIABLES:
          _startupData                    1000     2C
          RL_counter                      4000      2
```

Now we want to use the ROM library from our application. Our simple application is:

```
/* main application using ROM library: main.c */
#include "rl.h"

int cnt;

void main(void) {
  int i;

  for (i=0; i<100; i++) {
    cnt = RL_Count();
  }
}
```

After compiling this main.c we can link it with our ROM library:

```
LINK main.abs

NAMES  main.o romlib.lib startup.o ansi.lib END

SECTIONS
    MY_RAM = READ_WRITE  0x5000 TO 0x53FF;
    MY_ROM = READ_ONLY   0x6000 TO 0x6FFF;

PLACEMENT
    DEFAULT_ROM, ROM_VAR, STRINGS  INTO  MY_ROM;
    DEFAULT_RAM                    INTO  MY_RAM;
END

STACKSIZE  0x200
```

Note that depending on your CPU configuration and memory model you have to use another startup object file than ‘startup.o’ and another library than ‘ansi.lib’. Additionally you have to be careful to choose the right startup object file. For

efficiency reasons most of the startup files implemented in HLI are optimized for a specific target. To save ROM usage, they do not support ROM libraries in the startup code. As long as there is no Module Initialization needed, this is not a problem. But if we want to use the Module Initialization feature (as in our example), we use the ANSI-C implementation in the library directory (startup.c). Because this startup file may not be delivered in every target configuration, you have to compile this startup file 'startup.c' too.

After linking to main.abs, you get following map file (extract):

```
*****************************************************
STARTUP SECTION
-----------------------------------------------------
Entry point: 0x6000
Linker generated code (at 0x6000) before calling __Startup:
MOVE #0x2700, SR
JMP 0x61A0
_startupData is allocated at 600A and uses 48 Bytes

extern struct _tagStartup{
    unsigned flags                              0
    _PFunc   main                            603C (_main)
    unsigned dataPage                           0
    long     stackOffset                     5202
    int      nofZeroOuts                        1
    _Range   pZeroOut ->             5000       2
    long     toCopyDownBeg                   603A
    _PFunc   mInits   ->                     NONE
    void *   libInits ->                     1000
} _startupData;

*****************************************************
SEGMENT-ALLOCATION SECTION
-----------------------------------------------------
Segmentname           Size Type    From       To  Name

FUNCS                  184  R      603C     61BF  MY_ROM
COPY                     2  R      603A     603B  MY_ROM
STARTUP                 30  R      600A     6039  MY_ROM
_PRESTART                A  R      6000     6009  MY_ROM
SSTACK                 200 R/W     5002     5201  MY_RAM
DEFAULT_RAM              2 R/W     5000     5001  MY_RAM
*****************************************************
OBJECT-ALLOCATION SECTION
-----------------------------------------------------
```

```
Type:     Name:                         Address:  Size:
VECTOR
          value:        0                      0      4
          &_Startup                            4      4


MODULE:             -- main.o --
- PROCEDURES:
          main                              603C     26

- VARIABLES:
          cnt                               5000      2


MODULE:             -- X:\METROWERKS\DEMO\M68KC\rl.o --
- PROCEDURES:
          RL_Count                          102E     14 @

- VARIABLES:
          __startupData                     1000     2C @
          RL_counter                        4000      2 @


MODULE:             -- startup.o --
- PROCEDURES:
          ZeroOut                           6062     50
          CopyDown                          60B2     54
          ProcessStartupDesc                6142     3E
          HandleRomLibraries                6106     3C
          Start                             6180     20
          _Startup                          61A0     20

- VARIABLES:
          _startupData                      600A     30
```

Please note that objects linked from the ROM library (RL_Count, RL_counter) are marked with a ‘@’ in the OBJECT-ALLOCATION-SECTION. Again the linker has generated a startup descriptor at address 0x600A which points with field ‘libInits’ to the startup descriptor in our ROM library at address 0x1000.

Note that the main.abs does NOT include the code/data of the ROM library, thus they are NOT downloaded during downloading of main.abs, because they have to be downloaded (e.g. with a EEPROM) separately.

15

How To ...

How To Initialize the Vector Table

The vector table can be initialized in the assembly source file or in the linker parameter file. We recommend to initialize it in the prm file.

Initializing the Vector Table in the SmartLinker Prm File

Initializing the vector table from the prm file allows you to initialize single entries in the table. The user can decide if he wants to initialize all the entries in the vector table or not.

The labels or functions, which should be inserted in the vector table, must be implemented in the assembly source file. All these labels must be published otherwise they cannot be addressed in the linker prm file.

Example

```
XDEF IRQFunc, XIRQFunc, SWIFunc, OpCodeFunc, ResetFunc
DataSec: SECTION
Data:    DS.W 5  ; Each interrupt increments another element of the
table.
CodeSec: SECTION
; Implementation of the interrupt functions.
IRQFunc:
          LDAB #0
          BRA  int
XIRQFunc:
          LDAB #2
          BRA  int
SWIFunc:
          LDAB #4
          BRA  int
```

```
OpCodeFunc:
          LDAB #6
          BRA  int
ResetFunc:
          LDAB #8
          BRA  entry
int:
          LDX  #Data  ; Load address of symbol Data in X
          ABX         ; X <- address of the appropriate element in the
table
          INC  0, X   ; The table element is incremented
          RTI
entry:
          LDS #$AFE
loop:     BRA loop
```

NOTE The functions 'IRQFunc', 'XIRQFunc', 'SWIFunc', 'OpCodeFunc', 'ResetFunc' are published. This is required, because they are referenced in the linker prm file.

NOTE As the HC12 processor automatically pushes all registers on the stack on occurrence of an interrupt, the interrupt function do not need to save and restore the registers it is using.

NOTE All Interrupt functions must be terminated with an RTI instruction.

The vector table is initialized using the linker command VECTOR ADDRESS.

Example

```
LINK test.abs
NAMES
  test.o
END

SECTIONS
  MY_ROM  = READ_ONLY  0x0800 TO 0x08FF;
  MY_RAM  = READ_WRITE 0x0B00 TO 0x0CFF;
PLACEMENT
```

```
  DEFAULT_RAM          INTO MY_RAM;
  DEFAULT_ROM          INTO MY_ROM;
END

INIT ResetFunc
VECTOR ADDRESS 0xFFF2 IRQFunc
VECTOR ADDRESS 0xFFF4 XIRQFunc
VECTOR ADDRESS 0xFFF6 SWIFunc
VECTOR ADDRESS 0xFFF8 OpCodeFunc
VECTOR ADDRESS 0xFFFE ResetFunc
```

NOTE The statement ‘INIT ResetFunc’ defines the application entry point. Usually, this entry point is initialized with the same address as the reset vector.

NOTE The statement ‘VECTOR ADDRESS 0xFFF2 IRQFunc’ specifies that the address of function ‘IRQFunc’ should be written at address 0xFFF2.

Initializing the Vector Table in the Assembly Source File Using a Relocatable Section

Initializing the vector table in the assembly source file requires that all the entries in the table are initialized. Interrupts, which are not used, must be associated with a standard handler.

The labels or functions, which should be inserted in the vector table must be implemented in the assembler source file. The vector table can be defined in an assembly source file in an additional section containing constant variables.

Example

```
         XDEF ResetFunc
DataSec: SECTION
Data:    DS.W 5  ; Each interrupt increments an element of the table.
CodeSec: SECTION
; Implementation of the interrupt functions.
```

```
IRQFunc:
          LDAB #0
          BRA  int
XIRQFunc:
          LDAB #2
          BRA  int
SWIFunc:
          LDAB #4
          BRA  int
OpCodeFunc:
          LDAB #6
          BRA  int
ResetFunc:
          LDAB #8
          BRA  entry
DummyFunc:
          RTI
int:
          LDX  #Data
          ABX
          INC  0, X
          RTI
entry:
          LDS #$AFE
loop:     BRA loop

VectorTable:SECTION
; Definition of the vector table.
IRQInt:        DC.W IRQFunc
XIRQInt:       DC.W XIRQFunc
SWIInt:        DC.W SWIFunc
OpCodeInt:     DC.W OpCodeFunc
COPResetInt:   DC.W DummyFunc; No function attached to COP Reset.
ClMonResInt:   DC.W DummyFunc; No function attached to Clock
                         ; MonitorReset.
ResetInt     : DC.W ResetFunc
```

NOTE Each constant in the section ‘VectorTable’ is defined as a word (2 Byte constant), because the entries in the HC12 vector table are 16 bit wide.

NOTE In the previous example, the constant 'IRQInt' is initialized with the address of the label 'IRQFunc'.

NOTE In the previous example, the constant 'XIRQInt' is initialized with the address of the label 'XIRQFunc'.

NOTE All the labels specified as initialization value must be defined, published (using XDEF) or imported (using XREF) before the vector table section. No forward reference allowed in DC directive.

The section should now be placed at the expected address. This is performed in the linker parameter file.

Example

```
LINK test.abs
NAMES  test.o+ END

SECTIONS
  MY_ROM  = READ_ONLY  0x0800 TO 0x08FF;
  MY_RAM  = READ_WRITE 0x0A00 TO 0x0BFF;
/* Define the memory range for the vector table */
  Vector  = READ_ONLY  0xFFF2 TO 0xFFFF;
PLACEMENT
  DEFAULT_RAM        INTO MY_RAM;
  DEFAULT_RAM        INTO MY_ROM;
/* Place the section 'VectorTable' at the appropriated address. */
  VectorTable  INTO Vector;
END

INIT ResetFunc
```

NOTE The statement 'Vector = READ_ONLY 0xFFF2 TO 0xFFFF' defines the memory range for the vector table.

NOTE The statement 'VectorTable INTO Vector' specifies that the vector table should be loaded in the read only memory area Vector. This

means, the constant ‘IRQInt’ will be allocated at address 0xFFF2, the constant ‘XIRQInt’ will be allocated at address 0xFFF4, and so on. The constant ‘ResetInt’ will be allocated at address 0xFFFE.

NOTE The statement ‘NAMES test.o+ END’ switches smart linking OFF in the module test.o. If this statement is missing in the prm file, the vector table will not be linked with the application, because it is never referenced. The smart linker only links the referenced objects in the absolute file.

Initializing the Vector Table in the Assembly Source File Using an Absolute Section

Initializing the vector table in the assembly source file requires that all the entries in the table are initialized. Interrupts, which are not used, must be associated with a standard handler.

The labels or functions, which should be inserted in the vector table must be implemented in the assembly source file. The vector table can be defined in an assembly source file in an additional section containing constant variables.

Example

```
XDEF ResetFunc
DataSec: SECTION
Data:    DS.W 5  ; Each interrupt increments an element of the table.
CodeSec: SECTION
; Implementation of the interrupt functions.
IRQFunc:
          LDAB #0
          BRA  int
XIRQFunc:
          LDAB #2
          BRA  int
SWIFunc:
          LDAB #4
          BRA  int
OpCodeFunc:
          LDAB #6
```

```
          BRA  int
ResetFunc:
          LDAB #8
          BRA  entry
DummyFunc:
          RTI
int:
          LDX  #Data
          ABX
          INC  0, X
          RTI
entry:
          LDS #$AFE
loop:     BRA loop

               ORG $FFF2
; Definition of the vector table in an absolute section
; starting at address
;  $FFF2.
IRQInt:        DC.W IRQFunc
XIRQInt:       DC.W XIRQFunc
SWIInt:        DC.W SWIFunc
OpCodeInt:     DC.W OpCodeFunc
COPResetInt:   DC.W DummyFunc; No function attached to COP Reset.
ClMonResInt:   DC.W DummyFunc; No function attached to Clock
                             ; MonitorReset.
ResetInt     : DC.W ResetFunc
```

NOTE Each constant in the section ‘VectorTable’ is defined as a word (2 Byte constant), because the entry in the HC12 vector table are 16 bit wide.

NOTE In the previous example, the constant ‘IRQInt’ is initialized with the address of the label ‘IRQFunc’.

NOTE In the previous example, the constant ‘XIRQInt’ is initialized with the address of the label ‘XIRQFunc’.

NOTE All the labels specified as initialization value must be defined, published (using XDEF) or imported (using XREF) before the vector table section. No forward reference allowed in DC directive.

NOTE The statement ‘ORG $FFF2‘ specifies that the following section must start at address $FFF2.

Example

```
LINK test.abs
NAMES
  test.o+
END

SEGMENTS
  MY_ROM  = READ_ONLY  0x0800 TO 0x08FF;
  MY_RAM  = READ_WRITE 0x0A00 TO 0x0BFF;
PLACEMENT
  DEFAULT_RAM        INTO MY_RAM;
  DEFAULT_ROM        INTO MY_ROM;
END

INIT ResetFunc
```

NOTE The statement ‘NAMES test.o+ END’ switches smart linking OFF in the module test.o. If this statement is missing in the prm file, the vector table will not be linked with the application, because it is never referenced. The smart linker only links the referenced objects in the absolute file

16

Messages

Message Kinds

There are four kinds of messsages generated by the SmartLinker:

WARNING

A message will be printed and linking will continue. Warning messages are used to indicate possible programming errors to the user.

ERROR

A message will be printed and linking will be stopped. Error messages are used to indicate illegal syntax in the PRM file.

FATAL

A message will be printed and linking will be aborted. A fatal message indicates a severe error which anyway will stop the linker.

If the Linker prints out a message, the message contains a message code ('L' for Linker) and a four to five digit number. This number may be used to search very fast for the indicated message in the manual. The messages for the linker linking EL/DWARF object files are counted from L1000 to L1999. The messages for the linker linking HIWARE format object files are counted from L2000 to L2999. The messages common for both are counted until L999 and from L4000 to L4999.

All messages generated by the SmartLinker are documented in increasing number order for easy and fast retrieaval.

Each message also has a description and if available a short example with a possible solution or tips to fix a problem.

For each message the type of the message is also noted, e.g. [ERROR] indicates that the message is an error message.

Messages for Linking ELF/DWARF Object File Format

L1000 <command name> not found

[ERROR]

Description

This message is generated when a mandatory linker command is missing in the

PRM file.
<command name>: name of the command, which is not found in the PRM file.

The mandatory commands are:

- LINK, which contains the name of the absolute file to generate. If the option –O is specified on the command line this message is not generated when the command LINK is missing in the PRM file.
- NAMES, where the files building the application are enumerated.
- PLACEMENT, where at least the predefined section '.text' and '.data' must be associated with a memory range.

When the LINK command is missing the message will be:
`'LINK not found'`
When the NAMES command is missing the message will be:
`'NAMES not found'`
When the PLACEMENT command is missing the message will be:
`'PLACEMENT not found'`

Example

```
NAMES  fibo.o start12s.o ansis.lib END
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
PLACEMENT
    .text   INTO MY_ROM;
    .data   INTO MY_RAM;
    .stack  INTO MY_STK;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Insert the missing command in the PRM file.

Example

```
LINK   fibo.abs
NAMES  fibo.o start12s.o ansis.lib END
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
PLACEMENT
    .text   INTO MY_ROM;
    .data   INTO MY_RAM;
    .stack  INTO MY_STK;
END
```

```
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

L1001 <command name> multiply defined

[ERROR]

Description

This message is generated when a linker command, which is expected only once, is detected several times in the PRM file.

<command name>: name of the command, which is found twice in the PRM file.

The commands, which cannot be specified several times in a PRM file, are:

- LINK, which contains the name of the absolute file to generate.
- NAMES, where the files building the application are enumerated.
- SEGMENTS, where a name can be associated with a memory area.
- PLACEMENT, where the sections used in the application are assigned to a memory range.
- ENTRIES, where the objects, which should always be linked with the application, are enumerated.
- MAPFILE, where the information to be stored in the MAP file can be specified.
- MAIN, which defines the application main function.
- INIT, which defines the application entry point.
- STACKSIZE, which defines the size of the stack.
- STACKTOP, which defines the stack pointer initial value.
- OBJECT_ALLOCATION, where an absolute address or a section can be assigned to the objects in the application.
- LAYOUT, where the allocation order of the different objects can be defined.
- START_DATA, which defines the name of the startup structure.

When the LINK command is detected several times the message will be:

`'LINK multiply defined'`

Example

```
LINK    fibo.abs
NAMES   fibo.o start12s.o ansis.lib END
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
END
PLACEMENT
    .text   INTO MY_ROM;
    .data   INTO MY_RAM;
    .stack  INTO MY_STK;
END
LINK    fibo.abs
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Remove one of the duplicated command.

Example

```
LINK   fibo.abs
NAMES  fibo.o start12s.o ansis.lib END
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
END
PLACEMENT
    .text   INTO MY_ROM;
    .data   INTO MY_RAM;
    .stack  INTO MY_STK;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

L1003 Only a single SEGMENTS or SECTIONS block is allowed

[ERROR]

Description

The PRM file contains both a SECTIONS and a SEGMENTS block. The SECTIONS block is a synonym for the SEGMENTS block. It is supported for compatibility with old style HIWARE PRM file.

Example

```
LINK   fibo.abs
NAMES  fibo.o start12s.o ansis.lib END
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
END
SECTIONS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
PLACEMENT
    .text   INTO MY_ROM;
    .data   INTO MY_RAM;
    .stack  INTO MY_STK;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Remove either the SEGMENTS or the SECTIONS block.

L1004 <Token> expected

[ERROR]

Description

This message is generated, when the specified <Token> is missing at a position where it is expected.

<Token>: character or expression expected.

Example 1:

```
 SEGMENTS
   MY_RAM    = READ_WRITE 0x800 TO 0x8FF
               ALIGN [2TO 4, 4]
                            ^
 ERROR: : expected.
```

Tips

Insert the specified separator at the expected position.

L1005 Fill pattern will be truncated (>0xFF)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

This message is generated when the constant specified as fill pattern cannot be coded on a byte. The constant truncated to a byte value will be used as fill pattern.

Example

```
SEGMENTS
    MY_RAM = READ_WRITE 0x0800 TO 0x8FF FILL 0xA34;
END
```

Tips

To avoid this message, split the constant you specify into two byte constants.

Example

```
SEGMENTS
    MY_RAM = READ_WRITE 0x0800 TO 0x8FF FILL 0xA 0x34;
END
```

L1006 <Token> not allowed

[ERROR]

Description

This message is generated when a file name followed by a * is specified in a OBJECT_ALLOCATION or LAYOUT block. This is not possible, because a section is either a read only or a read write section. When all objects defined in a file are moved to a section, the destination section will contain both code and variable. This is logically not possible.

Example

```
OBJECT_ALLOCATION
    fibo.o:* INTO mySec;
          ^
```

```
ERROR: * not allowed
END
```

Tips

Move either all functions, or all variables, or all constants to the destination section.

Example

```
OBJECT_ALLOCATION
    fibo.o:CODE[*] INTO mySec;
END
```

L1007 <character> not allowed in file name (restriction)

[ERROR]

Description

A file name specified in the PRM file contains an illegal character.
<character>: list of characters, which are not allowed in a file name at the pointed position.
Following characters are not allowed in a file name:

- ‘:’, which is used as separator to specify a local object (function or variable) in a PRM file.
- ‘;’, which is used as delimiter for a command line in a LAYOUT or OBJECT_ALLOCATION block.
- ‘>’, which is used as separator to refer to object located in a section inside of a LAYOUT or OBJECT_ALLOCATION block.

We also recommend to avoid character ‘+’ and ‘-‘ in a file name. This may generate a problem when ‘+’ or ‘-‘ are used as suffix for a file name in the NAMES block.

Example

```
NAMES
    file:1.o;
        ^
ERROR: ':' or '>' not allowed in file name (restriction)
END
```

or

```
NAMES
    file1.o file>2.lib;
                ^
ERROR: ':' or '>' not allowed in file name (restriction)
END
```

Tips

Change the file name and avoid the illegal characters.

L1008 Only single object allowed at absolute address

[ERROR]

Description

Multiple objects are placed at an absolute address in an OBJECT_ALLOCATION block. Only single objects are allowed there.

Example

```
OBJECT_ALLOCATION
    var1 var2  AT 0x0800;
                 ^
ERROR: Only single object allowed at absolute address
END
```

or

```
OBJECT_ALLOCATION
     file.o:DATA[*] AT 0x900;
                       ^
ERROR: Only single object allowed at absolute address
END
```

Tips

Split the faulty command from the OBJECT_ALLOCATION command in several commands referring to single object.

Example

```
OBJECT_ALLOCATION
    var1  AT 0x0800;
    var2  AT 0x0802;
END
```

L1009 Segment Name <segment name> unknown

[ERROR]

Description

The segment specified in a PLACEMENT or LAYOUT command line was not previously defined in the SEGMENTS block.

<segment name>: name of the segment, which is not known.

Example

```
LINK   fibo.abs
NAMES  fibo.o start12s.o ansis.lib END
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
END
PLACEMENT
    .text   INTO ROM_AREA;
                       ^
ERROR: Segment Name ROM_AREA unknown
    .data   INTO MY_RAM;
    .stack  INTO MY_STK;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Define the requested segment names in the SEGMENTS block.

Example

```
LINK   fibo.abs
NAMES  fibo.o start12s.o ansis.lib END
SEGMENTS
    RAM_AREA = READ_WRITE 0x800 TO 0x80F;
    ROM_AREA = READ_ONLY  0x810 TO 0xAFF;
    STK_AREA = READ_WRITE 0xB00 TO 0xBFF;
END
PLACEMENT
    .text   INTO ROM_AREA;
    .data   INTO RAM_AREA;
    .stack  INTO STK_AREA;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

L1010 Section Name <section name> unknown

[ERROR]

Description

The section name specified in a command from the OBJECT_ALLOCATION block was not previously specified in the PLACEMENT block.
<section name>: name of the section, which is not known.

Example

```
LINK   fibo.abs
NAMES  fibo.o start12s.o ansis.lib END
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
END
PLACEMENT
    .text   INTO MY_ROM;
    .data   INTO MY_RAM;
    .stack  INTO MY_STK;
END
OBJECT_ALLOCATION
    fibo.o:DATA[*] IN dataSec;
                             ^
ERROR: Section Name dataSec unknown
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Specify the section in the PLACEMENT block.

Example

```
LINK   fibo.abs
NAMES  fibo.o start12s.o ansis.lib END
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
END
PLACEMENT
    .text            INTO MY_ROM;
    .data, dataSec   INTO MY_RAM;
    .stack           INTO MY_STK;
END
OBJECT_ALLOCATION
    fibo.o:DATA[*] IN dataSec;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

L1011 Incompatible segment qualifier: <qualifier1> in previous segment and <qualifier> in <segment name>

[ERROR]

Description

Two segments specified in the same command line from the PLACEMENT block are not defined with the same qualifier.

<qualifier1>: segment qualifier associated with the previous segment in the list. This qualifier may be READ_ONLY, READ_WRITE, NO_INIT, PAGED.

<qualifier2> segment qualifier associated with the current segment in the list. This qualifier may be READ_ONLY, READ_WRITE, NO_INIT, PAGED.

<segment name >: name of the current segment in the list.

Example

```
LINK   fibo.abs
NAMES  fibo.o start12s.o ansis.lib END
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    SEC_RAM= READ_WRITE 0x020 TO 0x02F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
END
PLACEMENT
    .data     INTO MY_RAM;
    .text     INTO MY_ROM, SEC_RAM;
```

```
                                  ^
ERROR: Incompatible segment qualifier: READ_ONLY in previous
segment and READ_WRITE in SEC_RAM
    .stack    INTO MY_STK;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Modify the qualifier associated with the specified segment.

Example

```
LINK   fibo.abs
NAMES  fibo.o start12s.o ansis.lib END
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    SEC_ROM= READ_ONLY  0x020 TO 0x02F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
END
PLACEMENT
    .data     INTO MY_RAM;
    .text     INTO MY_ROM, SEC_ROM;
    .stack    INTO MY_STK;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

L1012 Segment is not aligned on a <bytes> boundary

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Some targets (M-CORE, M68k) require aligned access for some objects.

Example (M-CORE)

All 4 byte accesses must be aligned to 4. According to the EABI, 8 byte doubles must be aligned to 8. But if a 8 byte structure only contains chars, then alignment is not needed.

Tips

Check whether the section contains objects which must be aligned.

L1013 Section is not aligned on a <bytes> boundary

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Some targets (M-CORE, M68k) require aligned access for some objects.

Example (M-CORE)

All 4 byte accesses must be aligned to 4. According to the EABI, 8 byte doubles must be aligned to 8. But if a 8 byte structure only contains chars, then alignment is not

needed.

Tips

Check whether the section contains objects which must be aligned.

L1015 No binary input file specified

[ERROR]

Description

No file names specified in the NAMES block.

Example

```
LINK    fibo.abs
NAMES END
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
END
PLACEMENT
    .text     INTO MY_ROM;
    .data     INTO MY_RAM;
    .stack    INTO MY_STK;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Specify at least a file name in the NAMES block.

L1016 File <filename> found twice

[DISABLE, INFORMATION, <u>WARNING</u>, ERROR]

Description

A file name is detected several times. The file may be specified in the NAMES block in the link parameter file or it may have been added by the option -Add.

<file name >: name of the file, which is detected twice.

Note that CodeWarrior is using the option -Add to add object files which are in the project. Therefore these files should not be mentioned in the prm file as well.

Example1§

```
LINK    fibo.abs
NAMES fibo.o startup.o fibo.o END
                              ^

WARNING L1016: File fibo.o found twice
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
END
PLACEMENT
```

```
    .text      INTO MY_ROM;
    .data      INTO MY_RAM;
    .stack     INTO MY_STK;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Remove the second occurrence of the specified file.

L1017 Section <Object/Section> in module <ModuleName> is incompatible with previous usages of this section

[DISABLE, INFORMATION, WARNING, ERROR]

Description

In the ELF object file format, two object files do contain the same section with incompatible modes. Sections containing code are for example incompatible with sections containing not initialized variable.

Example§

file1.c:

```
#pragma DATA_SEG MY_SEG
int i;
```

file2.asm

```
MY_SEG: SECTION
  NOP
```

file.prm

```
LINK  file.abs
NAMES file1.o file2.o .. END
...                          ^
```

Tips

Use different section names for different types of sections.

L1018 Checksum error <Description>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The checksum function has found a problem with the checksum configuration.

L1037 ***** Linking of <Linkparameterfile> failed ****

[ERROR]

Description

An error occurred in the linking process and the linking was interrupted and no output is written. The destination absolute file and the map file are killed by the Linker.

Tips

See the last error message for interpretation.

L1038 Success. Executable file written to <absfile>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The application was successfully linked and the specified application was created. When the linking fails, L1037 is issued. If the linking succeeds this message is issued, but as it is disabled by default, it is only visible if it was enabled with a command line option.

See also

Command line option -WmsgSi.

L1052 User requested stop

Description

[DISABLE, INFORMATION, WARNING, ERROR]

The user has pressed the stop button in the toolbar. The linker stops execution as soon as possible.

L1100 Segments <segment1 name> and <segment2 name> overlap

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Two segments defined in the PRM file overlap.

<segment1 name >: name of the first overlapping segment.

<segment2 name >: name of the second overlapping segment.

Example

```
^
Segments MY_RAM and MY_ROM overlap
LINK   fibo.abs
NAMES fibo.o startup.o END
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x805 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
END
PLACEMENT
    .text     INTO MY_ROM;
    .data     INTO MY_RAM;
    .stack    INTO MY_STK;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Modify the segment definition to remove the overlap.

Example

```
LINK   fibo.abs
NAMES fibo.o startup.o END
SEGMENTS
```

```
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
END
PLACEMENT
    .text     INTO MY_ROM;
    .data     INTO MY_RAM;
    .stack    INTO MY_STK;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

L1102 Out of allocation space in segment <segment name> at address <first address free>

[ERROR]

Description

The specified segment is not big enough to contain all objects from the sections placed in it.

<segment name>: is the name of the segment, which is too small.

<first address free>: is the first address free in this segment (i.e. the address following directly the last address used).

Example

In the following example, suppose the section '.data' contains a character variable and then a structure which size is 5 bytes.

```
^
Out of allocation space in segment MY_RAM at address 0x801
LINK   fibo.abs
NAMES fibo.o startup.o END
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x803;
    MY_ROM = READ_ONLY  0x805 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
END
PLACEMENT
    .text     INTO MY_ROM;
    .data     INTO MY_RAM;
    .stack    INTO MY_STK;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Set the end address of the specified segment to an higher value.

L1103 <section name> not specified in the PLACEMENT block

[ERROR]

Description

Indicates that one of the mandatory sections is not specified in the placement block. The sections, which must always be specified in the PLACEMENT block, are .text and .data.

Example

```
^
ERROR: .text not specified in the PLACEMENT block
LINK   fibo.abs
NAMES fibo.o startup.o END
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
END
PLACEMENT
    .init, .rodata     INTO MY_ROM;
    .data     INTO MY_RAM;
    .stack    INTO MY_STK;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Insert the missing section in the PLACEMENT block.

Note:

The sections DEFAULT_RAM is a synonym for .data and DEFAULT_ROM is a synonym for .text. These two sections name have been defined for compatibility with the old style HIWARE Linker.

L1104 Absolute object <Object Name> overlaps with segment <Segment Name>

[ERROR]

Description

An absolutely allocated object overlaps with a segment, where some section is allocated. This is not allowed, because this may cause multiple objects to be allocated at the same address.

Example

```
^
ERROR: Absolute object globInt overlaps with segment MY_RAM
LINK   fibo.abs
NAMES fibo.o startup.o END
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
```

```
END
PLACEMENT
    .text, .rodata     INTO MY_ROM;
    .data      INTO MY_RAM;
    .stack     INTO MY_STK;
END
OBJECT_ALLOCATION
      fiboCount AT 0x802;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Move the object to a free address.

Example

```
LINK   fibo.abs
NAMES fibo.o startup.o END
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
END
PLACEMENT
    .text      INTO MY_ROM;
    .data      INTO MY_RAM;
    .stack     INTO MY_STK;
END
OBJECT_ALLOCATION
     fiboCount AT 0xC00;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

Note:

An absolute object can also be placed in a segment, in which no sections are assigned.

Example

```
LINK   fibo.abs
NAMES fibo.o startup.o END
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
    ABS_MEM= READ_WRITE 0xC00 TO 0xC0F;
END
```

```
PLACEMENT
    .text     INTO MY_ROM;
    .data     INTO MY_RAM;
    .stack    INTO MY_STK;
END
OBJECT_ALLOCATION
      fiboCount AT 0xC00;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

L1105 Absolute object <object name> overlaps with another absolutely allocated object or with a vector

[ERROR]

Description

An absolutely allocated object overlaps with another absolute object or with a vector.

Example

```
^
ERROR: Absolute object globChar overlaps with another
absolutely allocated object or with a vector
LINK  fibo.abs
NAMES fibo.o startup.o END
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
END
PLACEMENT
    .text, .rodata     INTO MY_ROM;
    .data     INTO MY_RAM;
    .stack    INTO MY_STK;
END
OBJECT_ALLOCATION
      fiboCount AT 0xC02;
      counter   AT 0xC03;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Move the object to a free position.

L1106 <Object Name> not found

[ERROR]

Description

An object referenced in the PRM file or in the application is not found anywhere in

the application. This message is generated in following cases:

- An object moved to another section in the OBJECT_ALLOCATION block is not found anywhere in the application (WARNING).
- An object placed at an absolute address in the OBJECT_ALLOCATION block is not found anywhere in the application (ERROR).
- An object specified in a VECTOR or VECTOR ADDRESS command is not found anywhere in the application (ERROR).
- No start-up structure detected in the application (WARNING).
- An object (function or variable) referenced in another object is not found in the application (ERROR).
- An object (function or variable) specified in the ENTRIES block is not found in the application (ERROR).

Example

```
 ^
 ERROR: globInt not found
 LINK   fibo.abs
 NAMES fibo.o startup.o END

 SEGMENTS
     MY_RAM = READ_WRITE 0x800 TO 0x80F;
     MY_ROM = READ_ONLY  0x810 TO 0xAFF;
     MY_STK = READ_WRITE 0xB00 TO 0xBFF;
 END
 PLACEMENT
     .text, .rodata     INTO MY_ROM;
     .data      INTO MY_RAM;
     .stack     INTO MY_STK;
 END

 OBJECT_ALLOCATION
       globInt  AT 0xC02;
 END

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

The missing object must be implemented in one of the module building the application.

Make sure that your definition of the OBJPATH and GENPATH is correct and that the Linker uses the last version of the object files.

You can also check if all the binary files building the application are enumerated in the NAMES block.

L1107 <Object Name> not found

[DISABLE, INFORMATION, <u>WARNING</u>, ERROR]

Description

An object referenced in the PRM file or in the application is not found anywhere in the application. This message is generated in following cases:

- An object moved to another section in the OBJECT_ALLOCATION block is not found anywhere in the application (WARNING).
- An object placed at an absolute address in the OBJECT_ALLOCATION block is not found anywhere in the application (ERROR).
- An object specified in a VECTOR or VECTOR ADDRESS command is not found anywhere in the application (ERROR).
- No start-up structure detected in the application (WARNING).
- An object (function or variable) referenced in another object is not found in the application (ERROR).
- An object (function or variable) specified in the ENTRIES block is not found in the application (ERROR).

Example

```
 ^
 ERROR: globInt not found
 LINK   fibo.abs
 NAMES fibo.o startup.o END

 SEGMENTS
     MY_RAM = READ_WRITE 0x800 TO 0x80F;
     MY_ROM = READ_ONLY  0x810 TO 0xAFF;
     MY_STK = READ_WRITE 0xB00 TO 0xBFF;
 END
 PLACEMENT
     .text, .rodata     INTO MY_ROM;
     .data     INTO MY_RAM;
     .stack    INTO MY_STK;
 END

 OBJECT_ALLOCATION
       globInt  AT 0xC02;
 END

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

The missing object must be implemented in one of the module building the application.

Make sure that your definition of the OBJPATH and GENPATH is correct and that the Linker uses the last version of the object files.

You can also check if all the binary files building the application are enumerated in the NAMES block.

A missing _startupData is only issued if there is a non assembly object file or library

linked.

L1108 Initializing of Vector <Name> failed because of <Reason>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The linker can not initialize the named vector because of some target restrictions. Some processors do not imply any restrictions, while other do only allow the VECTOR's to point into a certain address range or have alignment constraints.

Tips

Try to allocate the interrupt function in a special segment and allocate this segment separately.

L1109 <Segment Name> appears twice in SEGMENTS block

[ERROR]

Description

A segment name is specified twice in a PRM file. This is not allowed. When this segment name is referenced in the PLACEMENT block, the Linker cannot detect which memory area is referenced.

Example

```
 LINK   fibo.abs
 NAMES fibo.o startup.o END

 SEGMENTS
     MY_RAM = READ_WRITE 0x800 TO 0x80F;
     MY_ROM = READ_ONLY  0x810 TO 0xAFF;
     MY_STK = READ_WRITE 0xB00 TO 0xBFF;
     MY_RAM = READ_WRITE 0xC00 TO 0xCFF;
            ^
 ERROR: MY_RAM appears twice in SEGMENTS block
 END
 PLACEMENT
     .text, .rodata     INTO MY_ROM;
     .data      INTO MY_RAM;
     .stack     INTO MY_STK;
 END

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Change one of the segment names, to generate unique segment names. If the same memory area is defined twice, you can remove one of the definitions.

L1110 <Segment Name> appears twice in PLACEMENT block

[ERROR]

Description

The specified segment appears twice in a PLACEMENT block, and in one of the PLACEMENT line, it is part of a segment list. A segment name may appear in several lines in the PLACEMENT block, if it is the only segment specified in the segment list. In that case the section lists specified in both PLACEMENT line are merged in one single list of sections, which are allocated in the specified segment.

Example

```
 LINK   fibo.abs
 NAMES fibo.o startup.o END

 SEGMENTS
     MY_RAM = READ_WRITE 0x800 TO 0x80F;
     MY_ROM = READ_ONLY  0x810 TO 0xAFF;
     MY_STK = READ_WRITE 0xB00 TO 0xBFF;
     ROM_2  = READ_ONLY  0x500 TO 0x7FF;
 END
 PLACEMENT
     .text, .rodata     INTO MY_ROM;
     .data              INTO MY_RAM;
     .stack             INTO MY_STK;
     codeSec1, codeSec2 INTO ROM_2, MY_ROM;
                                         ^
 ERROR: MY_ROM appears twice in PLACEMENT block
 END

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Remove one of the instance of the segment in the PLACEMENT block.

L1111 <Section Name> appears twice in PLACEMENT block

[ERROR]

Description

The specified section appears multiple times in a PLACEMENT block.

Example

```
 LINK   fibo.abs
 NAMES fibo.o startup.o END

 SEGMENTS
     MY_RAM = READ_WRITE 0x800 TO 0x80F;
     MY_ROM = READ_ONLY  0x810 TO 0xAFF;
     MY_STK = READ_WRITE 0xB00 TO 0xBFF;
     ROM_2  = READ_ONLY  0x500 TO 0x7FF;
 END
 PLACEMENT
     .text, .rodata     INTO MY_ROM;
```

```
    .data              INTO MY_RAM;
    .stack             INTO MY_STK;
    .text              INTO ROM_2;
         ^
ERROR: .text appears twice in PLACEMENT block
END

/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Remove one of the occurrence of the specified section from the PLACEMENT block.

L1112 The <Section name> section has segment type <Segment Qualifier> (illegal)

[ERROR]

Description

A section is placed in a segment, which has been defined with an incompatible qualifier. This message is generated in following cases:

- The section '.stack' is placed in a READ_ONLY segment.
- The section '.bss' is placed in a READ_ONLY segment.
- The section '.startData' is placed in a READ_WRITE, NO_INIT or PAGED segment.
- The section '.init' is placed in a READ_WRITE, NO_INIT or PAGED segment.
- The section '.copy' is placed in a READ_WRITE, NO_INIT or PAGED segment.
- The section '.text' is placed in a READ_WRITE, NO_INIT or PAGED segment.
- The section '.data' is placed in a READ_ONLY segment.

Example

```
^
ERROR: The .data section has segment type READ_ONLY (illegal)
LINK   fibo.abs
NAMES fibo.o startup.o END

SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
    ROM_2  = READ_ONLY  0x500 TO 0x7FF;
END
PLACEMENT
    .text, .rodata     INTO MY_ROM;
    .data              INTO ROM_2;
    .stack             INTO MY_STK;
END
```

```
/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Place the specified section in a segment, which has been defined with an appropriated qualifier.

Example

```
LINK   fibo.abs
NAMES fibo.o startup.o END

SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
    ROM_2  = READ_ONLY  0x500 TO 0x7FF;
END
PLACEMENT
    .text     INTO MY_ROM;
    .data     INTO MY_RAM;
    .stack    INTO MY_STK;
END
```

```
/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

L1113 The <Section name> section has segment type <Segment Qualifier> (illegal)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A section is placed in a segment, which has been defined with an incompatible qualifier. This message is generated in following cases:

- The section '.stack' is placed in a READ_ONLY segment.
- The section '.bss' is placed in a READ_ONLY segment.
- The section '.startData' is placed in a READ_WRITE, NO_INIT or PAGED segment.
- The section '.init' is placed in a READ_WRITE, NO_INIT or PAGED segment.
- The section '.copy' is placed in a READ_WRITE, NO_INIT or PAGED segment.
- The section '.text' is placed in a READ_WRITE, NO_INIT or PAGED segment.
- The section '.data' is placed in a READ_ONLY segment.

Example

```
^
ERROR: The .data section has segment type READ_ONLY (illegal)
LINK   fibo.abs
NAMES fibo.o startup.o END
```

```
SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
    ROM_2  = READ_ONLY  0x500 TO 0x7FF;
END
PLACEMENT
    .text, .rodata     INTO MY_ROM;
    .data              INTO ROM_2;
    .stack             INTO MY_STK;
END

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Place the specified section in a segment, which has been defined with an appropriated qualifier.

Example

```
LINK   fibo.abs
NAMES fibo.o startup.o END

SEGMENTS
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
    ROM_2  = READ_ONLY  0x500 TO 0x7FF;
END
PLACEMENT
    .text      INTO MY_ROM;
    .data      INTO MY_RAM;
    .stack     INTO MY_STK;
END

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

L1114 The <Section Name> section has segment type <Segment Qualifier> (initialization problem)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The specified section is loaded in a segment, which has been defined with the qualifier NO_INIT or PAGED. This may generate a problem because the section contains some initialized constants, which will not be initialized at application start-up. This message is generated in following cases:

- The section '.rodata' is placed in a NO_INIT or PAGED segment.
- The section '.rodata1' is placed in a NO_INIT or PAGED segment.

Example

```
 ^
 ERROR: The .rodata section has segment type NO_INIT
(initialization problem)
 LINK   fibo.abs
 NAMES fibo.o startup.o END

 SEGMENTS
     MY_RAM = READ_WRITE 0x800 TO 0x80F;
     MY_ROM = READ_ONLY  0x810 TO 0xAFF;
     MY_STK = READ_WRITE 0xB00 TO 0xBFF;
     RAM_2  = NO_INIT    0x500 TO 0x7FF;
 END
 PLACEMENT
     .text   INTO MY_ROM;
     .data   INTO MY_RAM;
     .stack  INTO MY_STK;
     .rodata INTO RAM_2;
 END

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Place the specified section in a segment defined with either the READ_ONLY or the READ_WRITE qualifier.

Example

```
 LINK   fibo.abs
 NAMES fibo.o startup.o END

 SEGMENTS
     MY_RAM = READ_WRITE 0x800 TO 0x80F;
     MY_ROM = READ_ONLY  0x810 TO 0xAFF;
     MY_STK = READ_WRITE 0xB00 TO 0xBFF;
     RAM_2  = NO_INIT    0x500 TO 0x7FF;
 END
 PLACEMENT
     .text     INTO MY_ROM;
     .data     INTO MY_RAM;
     .stack    INTO MY_STK;
     .rodata   INTO MY_ROM;
 END
```

```
/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

L1115 Function <Function Name> not found

[ERROR]

Description

The specified function is not found in the application. This message is generated in following cases:

- No main function available in the application. This function is not required for assembly application. For ANSI C application, if no main function is available in the application, it is the programmer responsibility to ensure that application start-up is performed correctly. Usually the main function is called 'main', but you can define your own main function using the linker command MAIN.
- No init function available in the application. The init function defines the entry point in the application. This function is required for ANSI C as well as for assembly application. Usually the init function is called '_Startup', but you can define your own init function using the linker command INIT.

Tips

Provide the application with the requested function.

L1116 Function <Function Name> not found

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The specified function is not found in the application. This message is generated in following cases:

- No main function available in the application. This function is not required for assembly application. For ANSI C application, if no main function is available in the application, it is the programmer responsibility to ensure that application startup is performed correctly. Usually the main function is called 'main', but you can define your own main function using the linker command MAIN.
- No init function available in the application. The init function defines the entry point in the application. This function is required for ANSI C as well as for assembly application. Usually the init function is called '_Startup', but you can define your own init function using the linker command INIT.

Tips

Provide the application with the requested function.

L1117 <Object Name> allocated at absolute address <Address> overlaps with sections placed in segment <Segment Name>

`[ERROR]`

Description

The specified absolutely allocated object is allocated inside of a segment, which is specified in the PLACEMENT block. This is not allowed, because the object may then overlap with object defined in the sections, which are placed in the specified segment.

An absolutely allocated object may be allocated inside of a segment, which do not appear in the PLACEMENT block.

Example

```
 ^
 ERROR: fiboCount allocated at absolute address 0x804 overlaps
with sections placed in segment MY_RAM
 LINK   fibo.abs
 NAMES fibo.o startup.o END

 SEGMENTS
     MY_RAM = READ_WRITE 0x800 TO 0x80F;
     MY_ROM = READ_ONLY  0x810 TO 0xAFF;
     MY_STK = READ_WRITE 0xB00 TO 0xBFF;
     RAM_2  = NO_INIT    0x500 TO 0x7FF;
 END
 PLACEMENT
     .text   INTO MY_ROM;
     .data   INTO MY_RAM;
     .stack  INTO MY_STK;
     .rodata INTO RAM_2;
 END

 OBJECT_ALLOCATION
     counter   AT 0x500;
     fiboCount AT 0x804;
 END
/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Move the absolutely allocated object to an unused address.

Example

```
 LINK   fibo.abs
 NAMES fibo.o startup.o END

 SEGMENTS
     MY_RAM = READ_WRITE 0x800 TO 0x80F;
     MY_ROM = READ_ONLY  0x810 TO 0xAFF;
     MY_STK = READ_WRITE 0xB00 TO 0xBFF;
     RAM_2  = NO_INIT    0x500 TO 0x7FF;
 END
 PLACEMENT
     .text     INTO MY_ROM;
     .data     INTO MY_RAM;
     .stack    INTO MY_STK;
```

```
     .rodata   INTO MY_ROM;
 END

 OBJECT_ALLOCATION
     counter   AT 0x500;
     fiboCount AT 0x404;
 END

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

L1118 Vector allocated at absolute address <Address> overlaps with another vector or an absolutely allocated object

[ERROR]

Description

A vector overlaps with an absolute object or with another vector.

Example

```
 ^
 ERROR: Vector allocated at absolute address 0xFFFE overlaps
with another vector or an absolutely allocated object
 LINK   fibo.abs
 NAMES fibo.o startup.o END

 SEGMENTS
     MY_RAM = READ_WRITE 0x800 TO 0x80F;
     MY_ROM = READ_ONLY  0x810 TO 0xAFF;
     MY_STK = READ_WRITE 0xB00 TO 0xBFF;
 END
 PLACEMENT
     .text, .rodata     INTO MY_ROM;
     .data     INTO MY_RAM;
     .stack    INTO MY_STK;
 END

 OBJECT_ALLOCATION
     counter   AT 0xFFFD;
 END

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Move the object or vector to a free position.

L1119 Vector allocated at absolute address <Address> overlaps with sections placed in segment <Segment Name>

[ERROR]

Description

The specified vector is allocated inside of a segment, which is specified in the PLACEMENT block. This is not allowed, because the vector may then overlap with object defined in the sections, which are placed in the specified segment.
A vector may be allocated inside of a segment, which do not appear in the PLACEMENT block.

Example

```
 ^
 ERROR: Vector allocated at absolute address 0xFFFE overlaps
with sections placed in segment ROM_2
 LINK   fibo.abs
 NAMES fibo.o startup.o END

 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
     MY_STK = READ_WRITE 0xB00  TO 0xBFF;
     ROM_2  = READ_ONLY  0xFF00 TO 0xFFFF;
 END
 PLACEMENT
     .text   INTO MY_ROM;
     .data   INTO MY_RAM;
     .stack  INTO MY_STK;
     .rodata INTO ROM_2;
 END

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Defined the specified segment outside of the vector table.

Example

```
 LINK   fibo.abs
 NAMES fibo.o startup.o END

 SEGMENTS
     MY_RAM = READ_WRITE 0x800 TO 0x80F;
     MY_ROM = READ_ONLY  0x810 TO 0xAFF;
     MY_STK = READ_WRITE 0xB00 TO 0xBFF;
     ROM_2  = READ_ONLY  0xC00 TO 0xCFF;
 END
 PLACEMENT
     .text   INTO MY_ROM;
     .data   INTO MY_RAM;
```

```
     .stack  INTO MY_STK;
     .rodata INTO ROM_2;
 END

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

L1120 Vector allocated at absolute address <Address> placed in segment <Segment Name>, which has not READ_ONLY qualifier

[ERROR]

Description

The specified vector is defined inside of a segment, which is not defined with the qualifier READ_ONLY. The vector table should be initialized at application loading time during the debugging phase. It should be burned into EPROM, when application development is terminated. For these reason, the vector table must always be located in a READ_ONLY memory area.

Example

```
 ^
 ERROR: Vector allocated at absolute address 0xFFFE placed in
segment RAM_2 which has not READ_ONLY qualifier
 LINK   fibo.abs
 NAMES fibo.o startup.o END

 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
     MY_STK = READ_WRITE 0xB00  TO 0xBFF;
     RAM_2  = READ_WRITE 0xFF00 TO 0xFFFF;
 END
 PLACEMENT
     .text   INTO MY_ROM;
     .data   INTO MY_RAM;
     .stack  INTO MY_STK;
 END

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Define the specified segment with the qualifier READ_ONLY.

L1121 Out of allocation space at address <Address> for .copy section

[ERROR]

Description

There is not enough memory available to store all the information about the initialized variables in the ‘.copy’ section.

Tips

Specify an higher end address for the segment, where the ‘.copy’ section is allocated.

L1122 Section .copy must be the last section in the section list

[ERROR]

Description

The section ‘.copy’ is specified in a section list from the PLACEMENT block, but it is not specified at the end of the list. As the size from this section cannot be evaluated before all initialization values are written, the .copy section must be the last section in a section list.

Example

```
  ERROR: Section .copy must be the last section in the section
list

 LINK   fibo.abs
 NAMES fibo.o startup.o END

 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
     MY_STK = READ_WRITE 0xB00  TO 0xBFF;
 END
 PLACEMENT
     .copy, .text   INTO MY_ROM;
     .data          INTO MY_RAM;
     .stack         INTO MY_STK;
 END

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Move the section .copy to the last position in the section list or define it on a separate PLACEMENT line in a separate segment.

Please note that .copy is also a synonym for COPY (e.g. used in HIWARE object file format prm files).

Example

```
 LINK   fibo.abs
 NAMES fibo.o startup.o END

 SEGMENTS
```

```
    MY_RAM = READ_WRITE 0x800 TO 0x80F;
    MY_ROM = READ_ONLY  0x810 TO 0xAFF;
    MY_STK = READ_WRITE 0xB00 TO 0xBFF;
    ROM_2  = READ_ONLY  0xC00 TO 0xDFF;
END
PLACEMENT
    .text   INTO MY_ROM;
    .data   INTO MY_RAM;
    .stack  INTO MY_STK;
    .copy   INTO ROM_2;
END

/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

L1123 Invalid range defined for segment <Segment Name>. End address must be bigger than start address

[ERROR]

Description

The memory range specified in the specified segment definition is not valid. The segment end address is smaller than the segment start address.

Example

```
LINK   fibo.abs
NAMES fibo.o startup.o END

SEGMENTS
    MY_RAM = READ_WRITE 0x800  TO 0x7FF;
                                     ^
ERROR: Invalid range defined for segment MY_RAM. End address
must be bigger than start address
    MY_ROM = READ_ONLY  0x810  TO 0xAFF;
    MY_STK = READ_WRITE 0xB00  TO 0xBFF;
END
PLACEMENT
   .text            INTO MY_ROM;
    .data           INTO MY_RAM;
    .stack          INTO MY_STK;
END

/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Change either the segment start or end address to define a valid memory range.

L1124 '+' or '-' should directly follow the file name

[ERROR]

Description

The ‘+’ or ‘-‘ suffix specified after a file name in the NAMES block does not directly follow the file name. There is at least a space between the file name and the suffix.

Example

```
LINK   fibo.abs
NAMES fibo.o + startup.o END
              ^
ERROR: '+' or '-' should directly follow the file name
SEGMENTS
    MY_RAM = READ_WRITE 0x800  TO 0x80F;
    MY_ROM = READ_ONLY  0x810  TO 0xAFF;
    MY_STK = READ_WRITE 0xB00  TO 0xBFF;
END
PLACEMENT
    .text          INTO MY_ROM;
    .data          INTO MY_RAM;
    .stack         INTO MY_STK;
END

/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Remove the superfluous space after the file NAME.

Example

```
LINK   fibo.abs
NAMES fibo.o+ startup.o END
SEGMENTS
    MY_RAM = READ_WRITE 0x800  TO 0x80F;
    MY_ROM = READ_ONLY  0x810  TO 0xAFF;
    MY_STK = READ_WRITE 0xB00  TO 0xBFF;
END
PLACEMENT
    .text          INTO MY_ROM;
    .data          INTO MY_RAM;
    .stack         INTO MY_STK;
END

/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

L1125 In small memory model, code and data must be located on bank 0. (StartAddr EndAddr)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The application has been assembled or compiled in small memory model and the memory area specified for some segment is not located on the first 64K (0x0000 to 0xFFFF).

This message is not issued for all processors.

Example

```
 ^
 ERROR: In small memory model, code and data must be located
on bank 0
 LINK   fibo.abs
 NAMES fibo.o startup.o END
 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x10810  TO 0x10AFF;
     MY_STK = READ_WRITE 0xB00  TO 0xBFF;
 END
 PLACEMENT
     .text          INTO MY_ROM;
     .data          INTO MY_RAM;
     .stack         INTO MY_STK;
 END

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

If some memory upper than 0xFFFF is required for the application, the application must be assembled or compiled using the medium memory model. If no memory upper than 0xFFFF is required, modify the memory range and place it on the first 64K of memory.

L1127 Placement located outside 16 bit area in small memory model in area StartAddr .. EndAddr

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The application has been assembled or compiled in small memory model and the memory area specified for some segment is not located on the first 64K (0x0000 to 0xFFFF).

This message is only issued for the HC12 and note that this message is disabled by default.

Example

```
 ^
 Warning: Placement located outside 16 bit area  in small memory
model in area 0x10810.. 0x10AFF
 LINK   fibo.abs
 NAMES fibo.o startup.o END
```

```
SEGMENTS
    MY_RAM = READ_WRITE 0x800  TO 0x80F;
    MY_ROM = READ_ONLY  0x10810  TO 0x10AFF;
    MY_STK = READ_WRITE 0xB00  TO 0xBFF;
END
PLACEMENT
    .text          INTO MY_ROM;
    .data          INTO MY_RAM;
    .stack         INTO MY_STK;
END

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

If some memory upper than 0xFFFF is required for the application, the application must be assembled or compiled using the medium memory model. If no memory upper than 0xFFFF is required, modify the memory range and place it on the first 64K of memory.

L1128 Cutting value <ItemName> from <FullValue> to <WrittenValue>

[DISABLED|INFORMATION|WARNING|ERROR]

Description

The linker does want to write a startup information entry which does not fit into the size available. The startup code defines the size available for an address, for example. If then larger addresses have to be written, this message is generated.

Example

For a startup code with 16 bits:

```
int i@0x12345678=7;
```

For the initialization of i, the linker has to encode the address of i (0x12345678) into two bytes. Obviously, the address has to be cutted, and the message is issued.

Tips

Check which kind of information did cause this message. Some startup codes do only support to initialize some part of the address space. This is especially the case when using small memory models and allocate variables in paged areas.

To avoid to generate (non working) initialization data, variables can be placed in a NO_INIT section.

The startup code can be adapted to support larger addresses.

Different memory models do have different limitations.

L1130 Section .checksum must be the last section in the section list

[DISABLED|INFORMATION|WARNING|ERROR]

Description

The section .checksum which will contains the linker generated checksum should itself not be considered for the checksum calculation. Therefore this section has to be

after all other sections.

Example

```
 LINK  chesksum.abs
 NAMES chesksum.o startup.o END

 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
 END
 PLACEMENT
     .checksum,.text INTO MY_ROM;
     .data           INTO MY_RAM;
 END
 STACKSIZE 0x60
/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Mention the .checksum section at the end of the section list or don’t mention it at all.

See also

Chapter CHECKSUM

L1200 Both STACKTOP and STACKSIZE defined

[ERROR]

Description

Both STACKTOP and STACKSIZE commands are specified in the PRM file. This is not allowed, because it generates an ambiguity on the definition of the stack.

Example

```
 LINK   fibo.abs
 NAMES fibo.o startup.o END

 STACKTOP 0xBFE
 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
 END
 PLACEMENT
     .text            INTO MY_ROM;
     .data           INTO MY_RAM;
 END
 STACKSIZE 0x60
/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Remove either the STACKTOP or the STACKSIZE command from the PRM file.

L1201 No stack defined

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The PRM file does not contains any stack definition. In that case it is the programmer responsibility to initialize the stack pointer inside of his application code. The stack can be defined in the PRM file in one of the following way:

- Trough the STACKTOP command in the PRM file.
- Trough the STACKSIZE command in the PRM file.
- Trough the specification of the section .stack in the placement block.

Example

```
 ^
WARNING: No stack defined
 LINK   fibo.abs
 NAMES fibo.o startup.o END

 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
 END
 PLACEMENT
     .text          INTO MY_ROM;
     .data          INTO MY_RAM;
 END
/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Define the stack in one of the three way specified above.

Note:

If the customer initializes the stack pointer inside of his source code, the initialization from the linker will be overwritten.

L1202 .stack cannot be allocated on more than one segment

[ERROR]

Description

The section .stack is specified on a PLACEMENT line, where several segments are enumerated. This is not allowed, because the memory area reserved for the stack must be contiguous and cannot be split over different memory range.

Example

```
 ^
 ERROR: stack cannot be allocated on more than one segment
 LINK   fibo.abs
 NAMES fibo.o startup.o END
```

```
 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
     STK_1 = READ_WRITE 0xB00  TO 0xBFF;
     STK_2 = READ_WRITE 0xD00  TO 0xDFF;
 END
 PLACEMENT
     .text          INTO MY_ROM;
     .data          INTO MY_RAM;
     .stack         INTO STK_1, STK_2;
 END

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Define a single segment with qualifier READ_WRITE or NO_INIT to allocate the stack.

L1203 STACKSIZE command defines a size of <Size> but .stack specifies a stacksize of <Size>

[ERROR]

Description

The stack is defined trough both a STACKSIZE command and placement of the .stack section in a READ_WRITE or NO_INIT segment, but the size specified in the STACKSIZE command is bigger than the size of the segment where the stack is allocated.

Example

```
 ^
 ERROR: STACKSIZE command defines a size of 0x120 but .stack
specifies a stacksize of 0x100
 LINK   fibo.abs
 NAMES fibo.o startup.o END
 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
     STK_1  = READ_WRITE 0xB00  TO 0xBFF;
 END
 PLACEMENT
     .text          INTO MY_ROM;
     .data          INTO MY_RAM;
     .stack         INTO STK_1;
 END

 STACKSIZE 0x120
/* Set reset vector on _Startup */
```

```
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

To avoid this message you can either adapt the size specified in the STACKSIZE command to fit into the segment where .stack is allocated or simply remove the command STACKSIZE.

If you remove the command STACKSIZE from the previous example, The linker will initialized a stack from 0x100 bytes. The stack pointer initial value will be set to 0xBFE.

Example

```
 LINK   fibo.abs
 NAMES fibo.o startup.o END
 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
     MY_STK = READ_WRITE 0xB00  TO 0xBFF;
 END
 PLACEMENT
     .text          INTO MY_ROM;
     .data          INTO MY_RAM;
     .stack         INTO MY_STK;
 END

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

If the size specified in a STACKSIZE command is smaller than the size of the segment where the section .stack is allocated, the stack pointer initial value will be evaluated as follows:

```
&lt;segment start address&gt; + <size in STACKSIZE> -
<Additional Byte Required by the processor.>
```

Example

```
 LINK   fibo.abs
 NAMES fibo.o startup.o END
 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
     MY_STK = READ_WRITE 0xB00  TO 0xBFF;
 END
 PLACEMENT
     .text          INTO MY_ROM;
     .data          INTO MY_RAM;
     .stack         INTO MY_STK;
 END
 STACKSIZE 0x60
/* Set reset vector on _Startup */
```

```
VECTOR ADDRESS 0xFFFE _Startup
```

In the previous example, the initial value for the stack pointer is evaluated as:

```
0xB00 + 0x60s -2 = 0xB5E
```

L1204 STACKTOP command defines an initial value of <stack top> but .stack specifies an initial value of <Initial Value>

[ERROR]

Description

The stack is defined trough both a STACKTOP command and placement of the .stack section in a READ_WRITE or NO_INIT segment, but the value specified in the STACKTOP command is bigger than the end address of the segment where the stack is allocated.

Example

```
 ^
 ERROR: STACKTOP command defines an initial value of 0xCFE but
.stack specifies an initial value of 0xBFF
 LINK   fibo.abs
 NAMES fibo.o startup.o END
 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
     STK_1  = READ_WRITE 0xB00  TO 0xBFF;
 END
 PLACEMENT
     .text          INTO MY_ROM;
     .data          INTO MY_RAM;
     .stack         INTO STK_1;
 END

 STACKTOP 0xCFE
/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

To avoid this message you can either adapt the address specified in the STACKTOP command to fit into the segment where .stack is allocated or simply remove the command STACKTOP.

If you remove the command STACKTOP from the previous example, the stack pointer initial value will be set to 0xBFE.

Example

```
 LINK   fibo.abs
 NAMES fibo.o startup.o END
 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
```

```
    MY_STK = READ_WRITE 0xB00  TO 0xBFF;
END
PLACEMENT
    .text           INTO MY_ROM;
    .data           INTO MY_RAM;
    .stack          INTO MY_STK;
END

/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

L1205 STACKTOP command incompatible with .stack being part of a list of sections

[ERROR]

Description

The stack is defined trough both a STACKTOP command and placement of the .stack section in a READ_WRITE or NO_INIT segment, but the .stack section is specified inside of a list of section in the PLACEMENT block.

Example

```
 ^
 ERROR: STACKTOP command incompatible with .stack being part
of a list of sections
 LINK   fibo.abs
 NAMES fibo.o startup.o END
 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
     STK_1  = READ_WRITE 0xB00  TO 0xBFF;
 END
 PLACEMENT
     .text           INTO MY_ROM;
     .data, .stack   INTO STK_1;
 END

 STACKTOP 0xBFE
/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Specify the .stack section in a placement line, where the stack alone is specified.

L1206 .stack overlaps with a segment which appear in the PLACEMENT block

[ERROR]

Description

The stack is defined trough the command STACKTOP, and the specified initial value

is inside of a segment, which is used in the PLACEMENT block.
This is not allowed, because the stack may overlap with some allocated objects.

Example

```
 ^
 ERROR: .stack overlaps with a segment which appear in the
PLACEMENT block
 LINK   fibo.abs
 NAMES fibo.o startup.o END
 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
     STK_1  = READ_WRITE 0xB00  TO 0xBFF;
 END
 PLACEMENT
     .text          INTO MY_ROM;
     .data          INTO STK_1;
 END

 STACKTOP 0xBFE
/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Define the stack initial value outside of all the segment specified in the PLACEMENT block.

Example

```
 LINK   fibo.abs
 NAMES fibo.o startup.o END
 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
     MY_STK = READ_WRITE 0xB00  TO 0xBFF;
 END
 PLACEMENT
     .text          INTO MY_ROM;
     .data          INTO MY_RAM;
 END
 STACKTOP 0xBFE

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

L1207 STACKSIZE command is missing

[ERROR]

Description

The stack is defined only trough the placement of the .stack section in a READ_WRITE or NO_INIT segment, but the .stack section is not alone in the section list. In this case a STACKSIZE command is required, to specify the size required for the stack by the application.

Example

```
 LINK   fibo.abs
 NAMES fibo.o startup.o END
 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
     STK_1  = READ_WRITE 0xB00  TO 0xBFF;
 END
 PLACEMENT
     .text          INTO MY_ROM;
     .data, .stack  INTO STK_1;
 END

/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Indicate the requested stack size in a STACKSIZE command.

L1301 Cannot open file <File Name>

[ERROR]

Description

The linker is not able to open the application map or absolute file or to open one of the binary file building the application.

Tips

If the abs or map file cannot be found, check if there is enough memory on the directory where you want to store the file. Check also if you have read/write access on this directory.

If the environment variable TEXTPATH is defined, the MAP file is stored in the first directory specified there, otherwise it is created in the directory, where the source file was detected.

If the environment variable ABSPATH is defined, the absolute file is stored in the first directory specified there, otherwise it is created in the directory, where the PRM file was detected.

If a binary file cannot be found, make sure the file really exist and his name is correctly spelled. Then check if your paths are defined correctly. The binary files must be located in one of the paths enumerated in the environment variable OBJPATH or GENPATH or in the working directory

L1302 File <File Name> not found

[ERROR]

Description

A file required during the link session cannot be found. This message is generated in following cases:

- The parameter file specified on the command line cannot be found.

Tips

Make sure the file really exist and his name is correctly spelled.
Then check if your paths are defined correctly. The PRM file must be located in one of the paths enumerated in the environment variable GENPATH or in the working directory.

L1303 <File Name> is not a valid ELF file

[ERROR]

Description

The specified file is not a valid ELF binary file. The linker is only able to link ELF binary files.

Tips

Check that you have compiled or assembled the specified file with the correct option to generate an ELF binary file.
Check if your paths are defined correctly. The binary files must be located in one of the paths enumerated in the environment variable OBJPATH or GENPATH or in the working directory.

L1305 <File Name> is not an ELF format object file (ELF object file expected)

[ERROR]

Description

The specified file is an old style HIWARE object file format binary file. The linker is only able to link ELF binary files.

Tips

Check that you have compiled or assembled the specified file with the correct option to generate an ELF binary file.
Check if your paths are defined correctly. The binary files must be located in one of the paths enumerated in the environment variable OBJPATH or GENPATH or in the working directory

L1309 Cannot open <File>

[ERROR]

Description

An input file of the linker is missed or the linker can’t open it.

Tips

Check your path environment settings in the “default.env” in your working directory.

L1400 Incompatible processor: <Processor Name> in previous files and <Processor Name> in current file

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The binary files building the application have been generated for different target processor. In this case, the linked code cannot be compatible.

Note that when this message is disabled, the produced absolute file may or may not work. The processor of the first read file is taken for the generation of fixups and similar entries. Because different processors define fixups and other topics differently, it is not predictable which combinations do really work.

Tips

Make sure you are compiling or assembling all your sources for the same processor. Check if your paths are defined correctly. The binary files must be located in one of the paths enumerated in the environment variable OBJPATH or GENPATH or in the working directory

L1401 Incompatible memory model: <Memory Model Name> in previous files and <Memory Model Name> in current file

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The binary files building the application have been generated for different memory model. In this case, the linked code cannot be compatible.

Tips

Make sure you are compiling or assembling all your sources in the same memory model.

Check if your paths are defined correctly. The binary files must be located in one of the paths enumerated in the environment variable OBJPATH or GENPATH or in the working directory.

This error can be moved to a warning to generate an abs file angering the problem. The problem may occur when linking object files of different vendors, because the memory model may not be correctly recognized.

When the memory model are compatible, this message can safely be switched off.

L1403 Unknown processor <Processor Constant>

[ERROR]

Description

The processor encoded in the binary object file is not a valid processor constant.

Tips

Check if your paths are defined correctly. The binary files must be located in one of the paths enumerated in the environment variable OBJPATH or GENPATH or in the working directory.

This message cannot be disabled because the meaning of fixups depends on the processor.

L1404 Unknown memory model <Memory Model Constant>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The memory model encoded in the binary object file is not a valid memory model for

the target processor.

Tips

Check if your paths are defined correctly. The binary files must be located in one of the paths enumerated in the environment variable OBJPATH or GENPATH or in the working directory.

This error can be moved to a warning to generate an abs file ignoring the problem. The problem may occur when linking object files of different vendors, because the memory model may not be correctly recognized.

When the memory model are compatible, this message can safely be switched off.

L1501 <Symbol Name> cannot be moved in section <Section Name> (invalid qualifier <Segment Qualifier>)

[ERROR]

Description

An invalid move operation has been detected from an object inside of a section, which appears only in the PRM file. In that case, the first object moved in a section determines the attribute associated with the section.

- If the object is a function, the section is supposed to be a code section,
- if the object is a constant, the section is supposed to be a constant section,
- otherwise, it is supposed to be a data section.

This message is generated:

- When a variable is moved in a section, which is placed in a READ_ONLY segment.
- When a function is moved in a section, which is placed in a READ_WRITE, NO_INIT or PAGED segment.

Example

```
ERROR: counter cannot be moved in section sec2 (invalid
qualifier READ_ONLY)
 LINK   fibo.abs
 NAMES fibo.o startup.o END
 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
     STK_1  = READ_WRITE 0xB00  TO 0xBFF;
 END
 PLACEMENT
     .text, sec2     INTO MY_ROM;
     .data           INTO MY_RAM;
      .stack         INTO STK_1;
 END

 OBJECT_ALLOCATION
   counter IN sec2;
 END
/* Set reset vector on _Startup */
```

```
VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Move the section in a segment with the required qualifier or remove the move command.

L1502 <Object Name> cannot be moved from section <Source Section Name> to section <Destination Section Name>

[ERROR]

Description

An invalid move operation has been detected from an object inside of a section, which appears also in a binary file.

This message is generated:

· When a variable is moved in a code or constant section

· When a function is moved in a data section or constant section.

Example

```
 ^
 ERROR: counter cannot be moved from section .data to section
.text
 LINK   fibo.abs
 NAMES fibo.o startup.o END
 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
     STK_1  = READ_WRITE 0xB00  TO 0xBFF;
 END
 PLACEMENT
     .text          INTO MY_ROM;
     .data          INTO MY_RAM;
      .stack        INTO STK_1;
 END

 OBJECT_ALLOCATION
   counter IN .text;
 END
/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Move the object in a section with the required attribute or remove the move command.

L1503 <Object Name> (from file <File Name>) cannot be moved from section <Source Section Name> to section <Destination Section Name>

[ERROR]

Description

An invalid move operation has been detected from objects defined in a binary file inside of a section.

This message is generated:

- When a variable is moved in a code or constant section
- When a function is moved in a data section or constant section.

Example

```
 ^
ERROR: counter (from file fibo.o) cannot be moved from section
.data to section .text
 LINK   fibo.abs
 NAMES fibo.o startup.o END
 SEGMENTS
     MY_RAM = READ_WRITE 0x800  TO 0x80F;
     MY_ROM = READ_ONLY  0x810  TO 0xAFF;
     STK_1  = READ_WRITE 0xB00  TO 0xBFF;
 END
 PLACEMENT
     .text          INTO MY_ROM;
     .data          INTO MY_RAM;
      .stack        INTO STK_1;
 END

 OBJECT_ALLOCATION
   fibo.o:[DATA] IN .text;
 END
/* Set reset vector on _Startup */
 VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Move the specified object in a section with the required attribute or remove the move command.

L1504 <Object Name> (from section <Section Name>) cannot be moved from section <Source Section Name> to section <Destination Section Name>

[ERROR]

Description

An invalid move operation has been detected from objects defined in a section inside of another section.

This message is generated:

- When a variable is moved in a code or constant section
- When a function is moved in a data section or constant section.

Example

```
^
```

```
ERROR: counter (from section .data) cannot be moved from
section .data to section .text
LINK   fibo.abs
NAMES fibo.o startup.o END
SEGMENTS
    MY_RAM = READ_WRITE 0x800  TO 0x80F;
    MY_ROM = READ_ONLY  0x810  TO 0xAFF;
    STK_1  = READ_WRITE 0xB00  TO 0xBFF;
END
PLACEMENT
    .text           INTO MY_ROM;
    .data           INTO MY_RAM;
    .stack          INTO STK_1;
END
OBJECT_ALLOCATION
  .data>[*] IN .text;
END
/* Set reset vector on _Startup */
VECTOR ADDRESS 0xFFFE _Startup
```

Tips

Move the specified object in a section with the required attribute or remove the move command.

L1600 main function detected in ROM library

[DISABLE, INFORMATION, <u>WARNING</u>, ERROR]

Description

A main function has been detected in a ROM library. As ROM libraries are not self executable applications, no main function is required there.

Tips

If the MAIN command is present in the PRM file, remove it.
If the application contains a function ‘main’, rename it.

L1601 startup function detected in ROM library

[DISABLE, INFORMATION, <u>WARNING</u>, ERROR]

Description

An application entry point has been detected in a ROM library. As ROM libraries are not self executable applications, no application entry point is required there.

Tips

If the INIT command is present in the PRM file, remove it.
If the application contains a function ‘_Startup’, rename it.

L1620 Bad digit in binary number

[ERROR]

Description

Syntax Error.

Illegal character in a binary number.

L1621 Bad digit in octal number

[ERROR]

Description

Syntax Error.
Illegal character in a octal number.

L1622 Bad digit in decimal number

[ERROR]

Description

Syntax Error.
Illegal character in a decimal number.

L1623 Number too big

[ERROR]

Description

Syntax Error.
An identifier in the link parameter file is limited to a length of 31 characters.

Tips

Reduce the length of the identifier.

L1624 Ident too long. Cut after 255 characters

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Syntax Error.
An identifier in the link parameter file is limited to a length of 255 characters. The identifier string is cut after that length.

Tips

Reduce the length of the identifier or move this message to a warning.

L1625 Comment not closed

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An ANSI-C comment (‘/* .... */’) was opened, but not closed.

Tips

Close the comment.

L1626 Unexpected end of file

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The end of file encountered and the scanner was involved in the inner scope of an expression or structure nesting. This is illegal.

Tips

Check the syntax of the link parameter file.

L1627 PRESTART command not supported, ignored

[DISABLE, INFORMATION, WARNING, ERROR]

Description

This message is issued by the linker when an ELF application is linked and the used link parameter file contains a PRESTART directive, which is not supported for ELF. The PRESTART command is only recognized from the parser to be able to skip it, but it is not implemented.

Tips

The prestart functionality can be achieved easily by adapting the startup code.

L1629 START_DATA command not supported yet

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The START_DATA command is already recognized from the parser, but not implemented yet.

Tips

Contact your vendor for the features of the next release.

L1631 HAS_BANKED_DATA not needed for ELF Object File Format

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The HAS_BANKED_DATA entry in the prm file is needed in the HIWARE file to define the size of pointers in the zero out and in the copy-down data structures. In the ELF format, the linker reads the sizes of the pointers from the DWARF2 debug info. When only DWARF1 is present, only one default pointer size per target is supported. The HAS_BANKED_DATA is completely ignored in the ELF Format.

L1632 Filename too long

[ERROR]

Description

A file name was longer as the limit for this file system.

Tips

As one filename can be longer than 250 characters under Win32 or most UNIX derivatives, the name did probably contain many paths. Try to use relative paths or use shorter path names.

L1633 Illegal Filename

[ERROR]

Description

A filename did contain an illegal character.

Tips

Win32 does not allow / \ : * ? “ < > | in filenames as they have a special semantic. Do use a different name instead.

L1634 Illegal Prestart

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The PRESTART link parameter file does not have correct parameters.

Tips

Prestart is not supported in ELF. Initialize your application in the startup code.

L1650 The encoding of <Object> in the special section .overlap was not recognized. The object is not overlapped

[DISABLE, INFORMATION, WARNING, ERROR]

Description

To overlap <Object> it must be known to which function this object belongs. The name of this function should be encoded into the object name. If the encoding is not correct, this message appears.

Tips

Do not use the sections .overlap and _OVERLAP for objects which should not be overlapped.
The compiler knows the section internally, so that these section names should only appear in the prm file and not in C sources.

L1651 The function <Function> of the overlap object <Object> was not found. The object is not overlapped

[DISABLE, INFORMATION, WARNING, ERROR]

Description

To overlap <Object> it must be known to which function this object belongs. The name of this function should be encoded into the object name. The encoding was recognized, but the corresponding function was not found or not linked.

Tips

Do not use the sections .overlap and _OVERLAP for objects which should not be overlapped.
The compiler knows the section internally, so that these section names should only appear in the prm file and not in C sources.

L1653 The object <Object> was not overlapped allocate

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The specified object is in the section .overlap and it's depending function was recognized. However, no root function did reached the function which corresponds to this object.

Tips

Add the name of the function to a OVERLAP_GROUP prm file entry.

L1654 <Object> was not marked as root for overlapping

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The <Object>, which may be a object file was not considered as root for the overlap analysis.

Tips

Add the name of all root functions into one or several OVERLAP_GROUP prm file entries.

L1655 Overlapping <Object> depends on itself

[DISABLE, INFORMATION, WARNING, ERROR]

Description

During the execution, the same function with overlapping objects must not be invoked twice. The linker has detect that one function depends on itself.

Example:

As recursion is not allowed with overlapping the following implementation is not only inefficient, it will even fail with overlapped variables.

```
int fibonacci(int i)  {
  return fibonacci(i-1)+fibonacci(i-2);
}
```

Tips

If the dynamic behavior of the function guarantees that no recursion takes place, ignore this warning. Otherwise change your code to avoid any recursion.

L1656 Overlapping <Object> depends on multiple roots

[DISABLE, INFORMATION, WARNING, ERROR]

Description

During the execution, the same function with overlapping objects must not be invoked twice. The linker has detect that one function depends on two root functions. This message is not issued for root objects.

Example:

In this example, the parameters of Mul are destroyed when Mul is invoked twice. As this happens only when the higher priority interrupt intercept the lower interrupt function, this bug is hard to catch with other tests. When both interrupt functions have the same priority, a OVERLAP_GROUP prm file entry should be used.

```
long l0,l1,l2,l3,l4,l5;
long Mul(long a, long b) {
  return a*b;
}
void interrupt 1 interrupt1(void) {
  l0=Mul(l1,l2);
}
void interrupt 2 interrupt2(void) {
  l3=Mul(l4,l5);
}
```

Tips

Check whether it is possible if the function is called twice at the same time. If so correct the code. Otherwise ignore this warning.
If the two roots cannot be called at the same time, a OVERLAP_GROUP prm file entry may save overlap space.

L1700 File <File Name> should contain DWARF information

[ERROR]

Description

The binary file, where the startup structure is defined does not contain any DWARF information. This is required, because the type of the startup structure is not fixed by the linker, but depends on the field and field position inside of the user defined structure.

Tips

Recompile the ANSI C file containing the definition of the startup structure and insert DWARF information there.

L1701 Start up data structure is empty

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The size of the user defined startup structure is 0 bytes.

Tips

Check if you really do not need any startup structure.
If a startup structure is available, check if the field name in the structure matches the name of the field expected by the linker.

L1702 Startup data structure field <name> is unknown

[DISABLE, INFORMATION, WARNING, ERROR]

Description

In the ELF object file format, the linker reads the debug information to build the startup data structures as the compiler expects them. Therefore no names in the startup structure should be changed. The linker did not find the information about the mentioned field, so no adoption takes place.

Tips

Check if the mentioned field exists in the startup data structure.
Check that all fields have the correct type.
If the startup information is not actually used, then it can be removed from the startup descriptor.

L1800 Read error in <File>

[ERROR]

Description

An error occurred while reading one of the ELF input object files. The object file is corrupt.

Tips

Recompile your sources. Contact your vendor, if the error appears again.

L1803 Out of memory in <Function Name>

[FATAL]

Description

There is not enough memory to allocate the internal structure required by the linker.

L1804 No Elf Section Header Table found in <File Name>

[ERROR]

Description

No section header table detected in the binary file.

Tips

Check if you are using the correct binary file.
Check if your paths are defined correctly. The binary files must be located in one of the paths enumerated in the environment variable OBJPATH or GENPATH or in the working directory

L1806 Elf file <File Name> appears to be corrupted

[ERROR]

Description

The specified binary file is not a valid ELF binary file.

Tips

Check if you are using the correct binary file.
Check if your paths are defined correctly. The binary files must be located in one of the paths enumerated in the environment variable OBJPATH or GENPATH or in the working directory

L1808 String overflow in <Function Name>, contact vendor

[ERROR]

Description

The section name detected in a section table is longer than 100 characters. This is an internal limit in this linker.

Tips

Ensure all the section names are smaller than 100 characters.

L1809 Section <Section Name> located in a segment with invalid qualifier

[ERROR]

Description

The attributes associated with a section, which is used in several binary file are not compatible. In one file, the section contains variables in the other it contains constants variables or code.

Tips

Check usage of the different sections over all the binary files. A specific section should always contain the same type of information, all over the project.

L1818 Symbol <Symbol Number> - <Symbol Name> duplicated in <First File Name> and <Second file Name>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The specified global symbol is defined in two different binary files.

Example

```
/* foo.h */
int i;
/* foo1.c */
#include "foo.h"
/* foo2.c */
#include "foo.h"
```

Tips

Rename the symbol defined in one on the specified files or check if a definition is present in a header file and included more than once (defined more than once).

L1820 Weak symbol <Symbol Name> duplicated in <First File Name> and <Second file Name>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The specified weak symbol is defined in two different binary files.

Tips

Rename the symbol defined in one on the specified files.

L1821 Symbol <id1> conflicts with <id2> in file <File> (same code)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A static symbol is defined twice in the same module.

Tips

Rename one of the symbols in the module.

L1822 Symbol <Symbol Name> in file <File Name> is undefined

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The specified symbol is referenced in the file specified, but is not defined anywhere in the application.

Tips

Check if there is no object file missing in the NAMES block and if you are using the correct binary file.

Check if your paths are defined correctly. The binary files must be located in one of the paths enumerated in the environment variable OBJPATH or GENPATH or in the working directory

L1823 External object <Symbol Name> in <File Name> created by

default

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Unresolved external.

The specified symbol is referenced in the file specified, but is not defined anywhere in the application, but an external declaration for this object is available in at least one of the binary file. The object is supposed to be defined in the first binary file where it is externally defined.

This is only valid for ANSI C applications.

In this case an external definition for a variable var looks like:

```
extern int var;
```

The definition of the corresponding variable looks like:

```
int var;
```

Tips

Define the specified symbol in one of the files building the application.

L1824 Invalid mark type for <Ident>

[ERROR]

Description

Internal error. The object file is corrupt.

Tips

Recompile your sources and contact your vendor if this leads to the same results.

L1826 Can't read file. <Filename> is a not an ELF library containing ELF objects (ELF objects expected)

[ERROR]

Description

The specified file is not a valid library. The linker is only able to link uniform binary files together (Not ELF and HIWARE mixed).

Tips

Recompile the source file to ELF object file format.

L1827 Symbol <Ident> has different size in <Filename> (<Size> bytes) and <Filename> (<Size> bytes)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An object was specified with different sizes in different object files.

This message is only issued if both sizes are specified in a object file. If one object file is contained in a library, L1828 is issued.

Example

```
a.h : exten char * buf;
      extern long intvar;
a.c : char buf[100];
      long intvar;
```

Tips

Check if all declarations and definitions of the named object match.
Recompile the source file to ELF object file format.
In C it is a recommended practise that the defining C file includes its own header file, even if this is not necessary to compile the C file. The compiler has only a chance to issue a warning about such cases if both the declarations in the header file and the definitions in the C file are read in one compilation.

See also

Message L1828

L1828 Library: Symbol <Ident> has different size in <Filename> (<Size> bytes) and <Filename> (<Size> bytes)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An object was specified with different sizes in different object files.
One of the two object files is contained in a library.
This message is only issued if one object file is contained in a library. If both sizes are specified in a object file, L1827 is issued.

Tips

Check if all declarations and definitions of the named object match.
Recompile the source file to ELF object file format.
In C it is a recommended practise that the defining C file includes its own header file, even if this is not necessary to compile the C file. The compiler has only a chance to issue a warning about such cases if both the declarations in the header file and the definitions in the C file are read in one compilation.

See also

Message L1827

L1829 Cannot resolve label 'Ident'

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The value of a label cannot be determined. This message may only be generated by assembly files.

Tips

Check the definition of the label.

L1902 <Cmd> command not supported

[DISABLE, INFORMATION, WARNING, ERROR]

Description

There are some command keywords in the link parameter file scanner, that are not yet implemented as commands. In this case, this message is issued.

Tips

See in your manual for the implemented commands.

L1903 Unexpected Symbol in Linkparameter file

[ERROR]

Description

Syntax error in link parameter file. An illegal character appeared.

Tips

It may accidentally happen that the link process is started with the name of the executable as file argument on command line instead of the link parameter file. In this case type the right file name.
If the file is really a link parameter file. Edit it and replace the invalid character or symbol.

L1905 Invalid section attribute for program header

[ERROR]

Description

Illegal object file.

Tips

Do recompile your sources. If this leads to the same results, contact your vendor for support.

L1906 Fixup out of buffer (<Obj> referenced at offset <Address>)

[ERROR]

Description

An illegal relocation of an object is detected in the object file <Object> at address <Address>. The type of the object is given in <objType>.

Tips

Check the relocation at that address. The offset may be out of range for this relocation type. If not it may be caused by a corrupt object file.

Recompile your sources and try to link again. If this leads to the same result, contact your vendor for support

L1907 Fixup overflow in <Object>, type <objType> at offset <Address>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An illegal relocation of an object is detected in the object file <Object> at address <Address>. The type of the object is given in <objType>.

Tips

Check the relocation at that address. The offset may be out of range for this relocation type. If not it may be caused by a corrupt object file.
Check if all objects are allocated in the correct area. Is the object correctly declared? This error might occur if the zero paged variables are allocated out of the zero page.

L1908 Fixup error in <Object>, type <objType> at offset <Address>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An illegal relocation of an object is detected in the object file <Object> at address <Address>. The type of the object is given in <objType>.

Tips

Check the relocation at that address. The offset may be out of range for this relocation type. If not it may be caused by a corrupt object file.
Recompile your sources and try to link again. If this leads to the same result, contact your vendor for support

L1910 Invalid section attribute for program header

[ERROR]

Description

A program header needs specific section attributes that have no sense to be changed.

Tips

The cause of the error is internal and may be caused by a corrupt object file.

Recompile your sources and try to link again. If this leads to the same result, contact your vendor for support

L1912 Object <obj> overlaps with another (last addr: <addr>, object address: <objadr>

[ERROR]

Description

The object with name <obj> overlaps with another object at address <addr>. The address of the object is given in <objadr>.

Tips

Do place one of the objects somewhere else.

L1913 Object Filler overlaps with something else

[ERROR]

Description

An object filler overlaps with another object this is not allowed.

L1914 Invalid object: <Object>

[ERROR]

Description

An object of unknown type is detected in an object file.

Tips

The cause of the error is internal and may be caused by a corrupt object file or incompatibility of the object formats.
Recompile your sources and try to link again. If this leads to the same result, contact your vendor for support or a new linker release, if the linker you are running is an older version that does not support the features of later compiler releases.

L1916 Section name <Section> is too long. Name is cut to 90 char-

acters length

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The length of a name is limited to 90 characters.

Tips

Rename the section and recompile your sources.

L1919 Duplicate definition of <Object> in library file(s) <File1> and/or <File2> discarded

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A definition of an object is duplicated in a library. (In object file <File1> and <File2>.

Tips

Rename one of the objects and recompile your sources.

L1921 Marking: too many nested procedure calls

[ERROR]

Description

The object file <name< is corrupt or your application.

Tips

Recompile your sources and try to link again. If this leads to the same result, contact your vendor for support.

L1922 File <filename> has DWARF data of different version, DWARF data may not be generated

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The files linked have different versions of the debug info sections (ELF/DWARF).

Tips

Recompile your sources with an unique version of output. See the compiler manual for the right option settings.

When linking object files of different vendors, this message might occur when the linker does not recognize the debug info in all object files.

It is also issued if some object files do not have debug info at all.

The generated absolute file may have some correct debug info, but probably not for all modules.

L1923 File <filename> has no DWARF debug info

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The mentioned file contains no recognized debug information. For the named object file the debugger will probably not show source files and other symbolic information. Its code can only be debugged on assembly level.

Tips

Metrowerks compilers contain an option to avoid the generation of debug information.
For other compiler, the generation of debug information must be explicitly specified. Check the compiler documentation.
The linker itself can also generate ROM libraries without debug information.

L1930 Unknown fixup type in <ident>, type <type>, at offset <offset>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The object file <name< is corrupt or your linker version does not support compiler instructions.

Tips

Recompile your sources and try to link again. If this leads to the same result, contact your vendor for support.

L1933 ELF: <details>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Warning while reading an ELF object file. The data in the file are not complete or consistent, but the ELF Linker can continue. <details> specifies the cause of the warning. Possible values are listed in the message L1934.

L1934 ELF: <details>

[ERROR]

Description

Error while reading an ELF object file. <details> specifies the cause of the error. Possible causes are:

- Cannot open <File> - See message L1309
- Read error in <File>
- Out of memory in <File> - See message L1803
- No Elf Section Header Table found in <File> - See message L1804
- Elf file <File> is corrupted - See message L1806
- String in '<File>' is too long - See message L1808
- Section '<File>' located in a segment with invalid qualifier. - See message L1809
- Programming language incompatible
- Incompatible memory model: <m1> in previous files and <m2> in current file - See message L1401
- Incompatible processor: <cpu1> in previous files and <cpu2> in current file - See message L1400
- String buffer overrun in <File>
- <File> is not a valid ELF file - See message L1303
- <File> is a HIWARE format object file (ELF object file expected) - See message L1305
- File <File> not found - See message L1302

- Requested section not found
- Program header not found
- Currently no file open
- Request is not valid
- Object <name> has an unknown type
- Fixup error: <cause>
- File is not a valid HIWARE library file
- File is not a valid ELF library file
- Elf file corrupted
- DWARF fixup incorrect: <cause>
- Internal

L1936 ELF output: <details>

[ERROR]

Description

Error in ELF. <details> specifies the cause of the error. Possible causes are:

- Cannot open <File> - See message L1309
- Out of memory in <File> - See message L1803
- Wrong file type for <action>
- Write error in <File>
- No Elf Section Header defined in <File>
- String buffer overrun in <File>
- Wrong section type
- Internal buffer overflow in <Function>
- All local symbols before the first global one
- Currently no file open
- Request is not valid
- Internal

L1937 LINK_INFO: <details>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The compiler does put with the #pragma LINK_INFO some information entries into the ELF file. This message is used if incompatible information entries exists.

Tips

Check the #pragma LINK_INFO in the compiler source.
This warning could indicate that some incompatible files are linked together.

L1951 Function <Function> is allocated inside of <Object> with offset <Offset>. Debugging may be affected

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The common code optimization of the linker has optimized one function. It is now allocated in the specified object. As both the function and the object are allocate at the same addresses, the debugger can not distinguish them. Be aware that the debug-

ger may display information for the wrong object.

Tips

If the two functions are identical per design, for example C++ inline functions, ignore the warning. If the function is very small, its influence might not be as large either. Check for large functions why your source does not only contain one instance.
In general be aware why the debugger steps suddenly into a completely different function.

L1952 Ident <name> too long. Cut after <size> characters

[DISABLE, INFORMATION, <u>WARNING</u>, ERROR]

Description

A very long identifier is truncated to the given length. Different as long identifiers with the same start until <Offset> may be mapped to the same name.

Tips

The linker supports more than 1000 character long names, so this message only occurs with really long names.
Very long names are generated by the C++ name mangling, as there class names occur as part of encoded parameter types of functions. If this is the reason, it might help to use shorter class names or to use less parameters, if possible.

L1970 Modifying code in function <function> at address <address> for ECALL

[DISABLE, INFORMATION, <u>WARNING</u>, ERROR]

Description

This message informs that the linker has modified the code for an ECALL instruction. That is that the linker has moved the ECALL instruction after the three following NOP instructions.

L1971 <Pattern> in function <function> at address <address> may be ECALL Pattern

[DISABLE, INFORMATION, <u>WARNING</u>, ERROR]

Description

The Linker has found a possible ECALL pattern at the given address. The Linker was not able to move this pattern.

Tips

The pattern may be produced by a data pattern. In this case check the code/data at the given address and map this message if this is ok.

L1972 <Pattern> in function <function> at address <address> looks like illegal ECALL

[DISABLE, INFORMATION, WARNING, <u>ERROR</u>]

Description

The Linker has found a possible ECALL pattern at the given address. The Linker was not able to move this pattern.

Tips

The pattern may be produced by a data pattern. In this case check the code/data at the given address and map this message if this is ok.

L1980 <Feature> not supported

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Linker does not support an used feature. This message is only used in rare circumstances, for example to show that some feature is not supported anymore (or not yet)

Tips

Check the documentation about this feature. Check why it was removed and if there are alternatives to use.

Messages for Linking HIWARE Object File Format

L2000 Segment <Segmentname> (for variables) should not be allocated in a READ_ONLY-section

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Variables must be allocated in RAM. The section <Segmentname>, containing variables was mapped in the PLACEMENT definition list of the link parameter file to a section that was defined as read only in the SECTIONS definition list. This is illegal.

Example (link parameter file)

```
.LINK bankdemo.abs
NAMES ansib.lib start12b.o  bankdemo.o END
SECTIONS
    MY_RAM =  READ_WRITE  0x0800 TO 0x0BFF;
    MY_ROM =  READ_ONLY   0xC000  TO 0xCFFF;
    VPAGE  =  READ_ONLY   0xD000  TO 0xFEFF;
    MY_PAGE = READ_ONLY 0x128000 TO 0x12AFFF;
PLACEMENT
    _PRESTART, STARTUP,
    ROM_VAR, STRINGS,
    NON_BANKED                   INTO  MY_ROM;
    DEFAULT_RAM                  INTO  MY_RAM;
    VPAGE_RAM                    INTO  VPAGE;
    MyPage,DEFAULT_ROM           INTO  MY_PAGE;
END
STACKSIZE 0x50
```

Example (header file)

```
#pragma DATA_SEG SHORT VPAGE
   int x[4];  /* 'x' is a variable, and can't therefore be */
              /*  allocated in a read only segment */
```

L2001 In link parameter file: segment <Segmentname> must always be present

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Some segments are required to be always present (mapped in the PLACEMENT definition list to an identifier defined in the SECTIONS definition list).

Example

DEFAULT_RAM and DEFAULT_ROM have always to be defined.

Tips

Use a template link parameter file, for your target, where these segments are always defined. Modify this file for your application. This way you avoid to write the same default settings for every application again and you will not forget to define the sec-

tions that have always to be present.

L2002 Library file <Library> (in module <Module>) incorrect: “cause”

[ERROR]

Description

Object file is corrupt.

Example (Cause)

“object tag incorrect” => The type tag of a linked object (VARIABLE, PROCEDURE,..) is incorrect.

Tips

Do compile your sources again. Contact Metrowerks support for help, if the error appears again.

L2003 Object file <Objfile> (<Cause>) incorrect

[ERROR]

Description

Object file is corrupt. (Equivalent message as L2002 for object files)

Tips

Do recompile the affected source file. Contact Metrowerks support for help, if the error appears again.

L2009 Out of allocation space in segment <segmentname> at address <address>

[ERROR]

Description

More address space allocated in segment <segmentname> than available. The address <address> given specifies the location, where the allocation failed.

L2008 Error in link parameter file

[ERROR]

Description

An error occurred while scanning the link parameter file. The message specifying the error was printed out as last message.

L2010 File not found: <Filename>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An input file (object file or absolute file) was not found.

Tips

Check your “default.env” path settings. Object files and absolute files opened for read are searched in the current directory or in the list of paths specified with the environment variables ‘OBJPATH’ and ‘GENPATH’.

L2011 File <filename> is not a valid HIWARE object file, absolute file

or library

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The file <filename> is expected to be a HIWARE object file, absolute file or library, because a file before in the NAMES list was a HIWARE format file. The linker started therefore to link the application in the HIWARE absolute file format .

Tips

You may have wished to link the application as ELF/DWARF executable, but the fist object file in the NAMES list found was detected to be a file in HIWARE format.
If you really intended to link an application in the HIWARE absolute format, replace the file <filename> by a valid HIWARE object file, absolute file or library.

L2014 User requested stop

Description

[DISABLE, INFORMATION, WARNING, ERROR]
The user has pressed the stop button in the toolbar. The linker stops execution as soon as possible.

L2015 Different type sizes in <ref_objfile> and <cur_objfile>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

In the HIWARE format, the size of many basic types (short, int, long, float, double, long double, default data pointer and default function pointer) are encoded into the object file. This message is issued if the linker detects two object files with different sizes. This may be caused an explicit setting of the types in some files only. Also for the assembler, the sizes cannot be modified. The linker is using the type sizes of the first specified object file. For the remaining, it does only issue this warning, if the size does not match. The sizes are used for the layout of the startup structure, the zero out and for the copy down information.

Tips

When the startup code object file is specified first, the startup structure sizes correspond to the startup code. Then differing informations in other object files do not matter and this warning can be ignored.
When this warning is generated by an assembly file, it can usually be ignored.
For C files, one has to be careful that functions are not getting incompatible when called from a different type setting than they are defined.

L2051 Restriction: library file <Library> (in module <Module>): <Cause>

[ERROR]

Description

There are some memory restrictions in the linker. This can be happen by the following causes:

Examples (Cause)

“too many objects”	Too many objects allocated.
“too many numeric initializer“	Too many initialized variables.
“too many address initializer“	Too many address initializer.

L2052 RESTRICTION: in object file <Objectfile>: <Cause>

[ERROR]

Description

Equivalent to message L2051, but for object files.

L2053 Module <Modulename> imported (needed for module-initialization?), but not present in list of objectfiles

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Only for linking MODULA-2.

The module <modulename> is in the import list of another module but not present in the list of object files, specified in the NAMES section of the link parameter file.

L2054 The symbolfiles of module <Modulename> (used from <User1> and <User2>) have different keys

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Only for linking MODULA-2.

With the link parameter file command CHECKKEYS ON, all keys of equal named imported modules are compared. CHECKKEYS ON is set by default. To switch of this check, write CHECKKEYS OFF in the .PRM file.

L2055 Function <functionname> (see link parameter file) not found

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An interrupt vector was mapped to the function with name <functionname> in the link parameter file. But a function with this name was not found in the modules linked.

L2056 Vector address <address> must fit wordsize

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An interrupt vector with word size is mapped to an odd address in the link parameter file.

L2057 Illegal file format (Reference to unknown object) in <objfile>

[ERROR]

Description

Older versions of HIWARE Compilers use -1 for unknown object.

This leads to inconsistency with the linker. Later versions of the HIWARE Compilers do avoid this. The error reported here is not a linker error, but a compiler error.

Tips

Do recompile your sources.

L2058 <objnum> referenced objects in <file>

[ERROR]

Description

Object file is corrupt: Too many referenced objects in file or the number of referenced objects is negative.

Tips

Do recompile the affected source file. Contact Metrowerks support for help, if the error appears again.

L2059 Error in map of <absfile>

[ERROR]

Description

Absolute file as input for ROM library is corrupt (Its number of modules is invalid).

Tips

Decode the absolute file. If this works and the number of modules contained is correct, contact Metrowerks support otherwise do rebuild the absolute file. Contact its distributor support for help, if this is not possible (absolute file from other party).

L2060 Too many (<objnum>) objects in library <library>

[ERROR]

Description

Number of objects in library exceeds maximum limit.
The actual value for the maximum depends on the linker version. The 32 Bit linker version allows more than 500’000’000 objects in one library. Old 16 bit linker versions did have a limit of 8000 objects.

Tips

Cause can be a corrupt library. Do divide your library in sub-libraries if the count is correct and this large.

L2061 <filename> followed by '-'/'+', but not a library or program module

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The plus sign after a file name in the NAMES section disables smart linking for the specified file. A minus sign specified after an absolute file name takes it out from application startup.

L2062 <object> found twice with different size (in '<module1>'-><objsize1> and in '<module2>'-><objsize2>)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Naming conflict or duplicated definition with different attributes in the application. Two objects where defined with the same name, but with different sizes.

L2063 <symbol> twice exported (module <module1> and module <module2>)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The object <symbol> has been implemented and exported from two different modules.

Tips

Review your module structure design. Remove one of the objects, if they refer to the same context. Rename one of the objects if both of them are used in different contexts.

L2064 Required system object <objectname> not found

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An object absolutely required by the linker is missing.

Example

'_Startup' is such an object.

Tips

The entry point of the application must exist in order to link correctly. It's default name is _Startup.
This name can be configured by the link parameter file entry INIT.
E.g.
...
INIT MyEntryPoint
..
Probably you forgot to specify the startup module as one of the files in the NAMES section. _Startup is thought to be defined in the startup module. Another reason can be name mangling with C++: The names of functions are encoded with the types in the object file, e.g. 'void Startup(void)' is encoded as 'Startup__Fv'. Either use 'extern "C"' for such cases or use the mangled name in the linker parameter file.

L2065 No module exports with name <objectname>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An object absolutely required by the linker is not exported.

Example

'__Startup' is such an object.

Tips

Probably you forgot to specify the startup module as one of the files in the NAMES section. __Startup is thought to be defined in the startup module.

L2066 Variable "_startupData" not found, linker prepares no startup

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The '_startupData' is a structure (C-struct) containing all information read out from the Startup function as:

- Top level procedure of user program
- Initial value of the stack pointer
- Number of zero out ranges
- Vector of ranges with nofZeroOuts elements
- Rom-address where copydown-data begins
- Number of library startup descriptors
- Vector of pointers to library startup descriptors
- Number of init functions for C++ constructors
- Vector of function pointers to init functions for C++ constructors

Without this structure, no startup can be prepared.

Tips

Probably you forgot to specify the startup module as one of the files in the NAMES section. __Startup is thought to be defined in the startup module.

If you do not want that the C startup code does perform any operation, you can safely disable this message.

Having no startup code is common in assembly programming, but it is an advanced feature with C programming as all of the above C/C++ features will not automatically work anymore.

L2067 Variable "_startupData" found, but not exported

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The startup data has been found, but is not exported.

Tips

See L2064.

L2068 <objname> (in ENTRIES link parameter file) not found

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Object name in the ENTRIES section was not found. In ENTRIES all objects are listed that are linked in any case (referenced or not by other objects). <objname> was not found in any module.

Tips

Check out, if the name in the ENTRIES section was written correctly.

L2069 The segment "COPY" must not cross sections

[ERROR]

Description

The COPY segment must be placed in one section. This is not the case here.

L2070 The segment STRINGS crosses the page boundary

[ERROR]

Description

The HC16 does not allow, the STRINGS section to cross page boundary.

L2071 Fixup Error: Reference to non linked object (<objname>)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An object was referenced, but not linked. This error may be caused by a modified dependency with DEPENDENCY in the prm file or by a wrong compiler/assembler specified dependency.

Tips

Use DEPENDENCY ADDUSE instead of DEPENDENCY USES. If the compiler/assembler did generate the missing dependencies, try to rebuild the application.

See also

Link Parameter File Command DEPENDENCY

L2072 8 bit branch (from address <address>) out of range (-128 <= <offset> <= 127)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

8 bit branch from address <address> out of range.

Tips

If the source file of this branch is an assembly file, take a look at the branch at address <address>. Rewrite to code correctly.
Some compilers do assume that functions compiled in the same segment and defined close together get allocated in the same order.
When in the link parameter file PLACEMENT such a segment is splitted up into several sections, then larger gaps can be generated:

```
SECTIONS
  ROM1 = READ_ONLY 0x4000 TO 0x4FFF;
  ROM2 = READ_ONLY 0x8000 TO 0x8FFF;
PLACEMENT
  FUNCTIONS INTO ROM1, ROM2;
```

If this causes your problem, either recompile your sources with this optimization switched off (see the compiler manual for the correct option) or splitup the functions by hand into different segments, which are assigned to one section only.

L2073 11 bit branch out of range (-2048 <= <offset> <= 2047)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

11 bit branch out of range.

Tips

If the source file of this branch is an assembly file, take a look at the branch at address <address>. Rewrite to code correctly.
Some compilers do assume that functions compiled in the same segment and defined close together get allocated in the same order.

When in the link parameter file PLACEMENT such a segment is splitted up into several sections, then larger gaps can be generated:

```
SECTIONS
  ROM1 = READ_ONLY 0x4000 TO 0x4FFF;
  ROM2 = READ_ONLY 0x8000 TO 0x8FFF;
PLACEMENT
  FUNCTIONS INTO ROM1, ROM2;
```

If this causes your problem, either recompile your sources with this optimization switched off (see the compiler manual for the correct option) or splitup the functions by hand into different segments, which are assigned to one section only.

L2074 16 bit branch out of range (-32768 <= <offset> <= 32767)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

16 bit branch out of range.

Tips

If the source file of this branch is an assembly file, take a look at the branch at address <address>. Rewrite to code correctly.
Some compilers do assume that functions compiled in the same segment and defined close together get allocated in the same order.
When in the link parameter file PLACEMENT such a segment is splitted up into several sections, then larger gaps can be generated:

```
SECTIONS
  ROM1 = READ_ONLY 0x4000 TO 0x4FFF;
  ROM2 = READ_ONLY 0x8000 TO 0x8FFF;
PLACEMENT
  FUNCTIONS INTO ROM1, ROM2;
```

If this causes your problem, either recompile your sources with this optimization switched off (see the compiler manual for the correct option) or splitup the functions by hand into different segments, which are assigned to one section only.

L2075 8 bit index out of range (<index> for <objname>)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Offset to index register is out of range.

Tips

If the source file of this branch is an assembly file, take a look at the branch at address <address>. Rewrite to code correctly.
Some compilers do assume that functions compiled in the same segment and defined close together get allocated in the same order.
When in the link parameter file PLACEMENT such a segment is splitted up into several sections, then larger gaps can be generated:

```
SECTIONS
  ROM1 = READ_ONLY 0x4000 TO 0x4FFF;
  ROM2 = READ_ONLY 0x8000 TO 0x8FFF;
```

```
PLACEMENT
  FUNCTIONS INTO ROM1, ROM2;
```

If this causes your problem, either recompile your sources with this optimization switched off (see the compiler manual for the correct option) or splitup the functions by hand into different segments, which are assigned to one section only.

L2076 Jump crossing page boundary

[ERROR]

Description

A jump is crossing page boundary.

Tips

If the source file of this branch is an assembly file, take a look at the branch at address <address>. Rewrite to code correctly.
Some compilers do assume that functions compiled in the same segment and defined close together get allocated in the same order.
When in the link parameter file PLACEMENT such a segment is splitted up into several sections, then larger gaps can be generated:

```
SECTIONS
  ROM1 = READ_ONLY 0x4000 TO 0x4FFF;
  ROM2 = READ_ONLY 0x8000 TO 0x8FFF;
PLACEMENT
  FUNCTIONS INTO ROM1, ROM2;
```

If this causes your problem, either recompile your sources with this optimization switched off (see the compiler manual for the correct option) or splitup the functions by hand into different segments, which are assigned to one section only.

L2077 16-bit index out of range (<index> for <objname>)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

16 bit index out of range.

Tips

If the source file of this branch is an assembly file, take a look at the branch at address <address>. Rewrite to code correctly.
Some compilers do assume that functions compiled in the same segment and defined close together get allocated in the same order.
When in the link parameter file PLACEMENT such a segment is splitted up into several sections, then larger gaps can be generated:

```
SECTIONS
  ROM1 = READ_ONLY 0x4000 TO 0x4FFF;
  ROM2 = READ_ONLY 0x8000 TO 0x8FFF;
PLACEMENT
  FUNCTIONS INTO ROM1, ROM2;
```

If this causes your problem, either recompile your sources with this optimization switched off (see the compiler manual for the correct option) or splitup the functions by hand into different segments, which are assigned to one section only.

L2078 5 bit offset out of range (-16 <= <offset> <= 15)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

5 bit offset out of range.

Tips

If the source file of this branch is an assembly file, take a look at the branch at address <address>. Rewrite to code correctly.
Some compilers do assume that functions compiled in the same segment and defined close together get allocated in the same order.
When in the link parameter file PLACEMENT such a segment is splitted up into several sections, then larger gaps can be generated:

```
SECTIONS
  ROM1 = READ_ONLY 0x4000 TO 0x4FFF;
  ROM2 = READ_ONLY 0x8000 TO 0x8FFF;
PLACEMENT
  FUNCTIONS INTO ROM1, ROM2;
```

If this causes your problem, either recompile your sources with this optimization switched off (see the compiler manual for the correct option) or splitup the functions by hand into different segments, which are assigned to one section only.

L2079 9 bit offset out of range (-256 <= <offset> <= 255) in <object> with offset <offset> to <object>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

9 bit offset out of range.

Tips

If the source file of this branch is an assembly file, take a look at the branch at address <address>. Rewrite to code correctly.
Some compilers do assume that functions compiled in the same segment and defined close together get allocated in the same order.
When in the link parameter file PLACEMENT such a segment is splitted up into several sections, then larger gaps can be generated:

```
SECTIONS
  ROM1 = READ_ONLY 0x4000 TO 0x4FFF;
  ROM2 = READ_ONLY 0x8000 TO 0x8FFF;
PLACEMENT
  FUNCTIONS INTO ROM1, ROM2;
```

If this causes your problem, either recompile your sources with this optimization switched off (see the compiler manual for the correct option) or split the functions by hand into different segments, which are assigned to one section only.

L2080 10 bit offset out of range (0 <= <offset> <= 1023)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

10 bit offset out of range.

Tips

If the source file of this branch is an assembly file, take a look at the branch at address <address>. Rewrite to code correctly.

Some compilers do assume that functions compiled in the same segment and defined close together get allocated in the same order.

When in the link parameter file PLACEMENT such a segment is splitted up into several sections, then larger gaps can be generated:

```
SECTIONS
  ROM1 = READ_ONLY 0x4000 TO 0x4FFF;
  ROM2 = READ_ONLY 0x8000 TO 0x8FFF;
PLACEMENT
  FUNCTIONS INTO ROM1, ROM2;
```

If this causes your problem, either recompile your sources with this optimization switched off (see the compiler manual for the correct option) or splitup the functions by hand into different segments, which are assigned to one section only.

L2081 Illegal allocation of BIT segment ('<objname>':0x<address>..0x<endaddress> => 0x20..0x3F, 0x400..0x43F)

[ERROR]

Description

Illegal allocation of BIT segment.

L2082 4 bit offset out of range (-7 <= <offset> <= 15)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

4 bit offset out of range.

Tips

If the source file of this branch is an assembly file, take a look at the branch at address <address>. Rewrite to code correctly.

Some compilers do assume that functions compiled in the same segment and defined close together get allocated in the same order.

When in the link parameter file PLACEMENT such a segment is splitted up into several sections, then larger gaps can be generated:

```
SECTIONS
  ROM1 = READ_ONLY 0x4000 TO 0x4FFF;
  ROM2 = READ_ONLY 0x8000 TO 0x8FFF;
PLACEMENT
  FUNCTIONS INTO ROM1, ROM2;
```

If this causes your problem, either recompile your sources with this optimization switched off (see the compiler manual for the correct option) or splitup the functions by hand into different segments, which are assigned to one section only.

L2083 11 bit offset out of range (-2048 <= <offset> <= 2047)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

11 bit offset out of range.

Tips

If the source file of this branch is an assembly file, take a look at the branch at address <address>. Rewrite to code correctly.
Some compilers do assume that functions compiled in the same segment and defined close together get allocated in the same order.
When in the link parameter file PLACEMENT such a segment is splitted up into several sections, then larger gaps can be generated:

```
SECTIONS
  ROM1 = READ_ONLY 0x4000 TO 0x4FFF;
  ROM2 = READ_ONLY 0x8000 TO 0x8FFF;
PLACEMENT
  FUNCTIONS INTO ROM1, ROM2;
```

If this causes your problem, either recompile your sources with this optimization switched off (see the compiler manual for the correct option) or splitup the functions by hand into different segments, which are assigned to one section only.

L2084 Can't solve reference to object <name>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Illegal or incompatible object file format. The reference to object <name> can't be solved.

Tips

Recompile the sources. If the result remains the same, contact Metrowerks support for help.

L2085 Can't solve reference to internal object

[ERROR]

Description

Illegal or incompatible object file format. The reference to object <name> can't be solved.

Tips

Recompile the sources. If the result remains the same, contact Metrowerks support for help.

L2086 Cannot switch to segment <segName>. (Offset to big)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Can't switch to segment <segname>. The offset is too big.

L2087 Object file position error in <objname>

[ERROR]

Description

Object file is corrupt.

Tips
Recompile the sources. If the result remains the same, contact Metrowerks support for help.

L2088 Procedure <funcname> not correctly defined

[ERROR]

Description
The named function was not defined. This error message does occur for example in the case of an used undefined static function.

Tips
Check if this static function is defined.

L2089 Internal: Code size of <objname> incorrect (<data> <objsize>)

[ERROR]

Description
Illegal object file format. The compiler or the assembler have produced a corrupt object file or the file has been corrupted after creation.

Tips
Do recompile your sources. If recompiling leads to the same results, contact Metrowerks support for help.

L2090 Internal: Failed to write procedures for <modulename>

[ERROR]

Description
Illegal object file format. The compiler or the assembler have produced a corrupt object file or the file has been corrupted after creation.

Tips
Do recompile your sources. If recompiling leads to the same results, contact Metrowerks support for help.

L2091 Data allocated in ROM can't exceed 32KByte

[DISABLE, INFORMATION, WARNING, ERROR]

Description
An object allocated in ROM is bigger that 32K. This is not allowed. The object won't be allocated.

L2092 Allocation of object <objname> failed

[DISABLE, INFORMATION, WARNING, ERROR]

Description
This error does occur when reserved linker segment names were used as identifier or when functions (code) should be placed into a non READ_ONLY segment.

Tips
Do not use reserved names for objects.
Check that all you code is placed into READ_ONLY segments.

L2093 Variable <varname> (objectfile <objfile>) appears in module <module1> and in module <module2>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A variable is defined twice (placed at different locations) in different modules.

L2094 Object <varname> (objectfile <objfile>) appears in module <module1> and in module <module2>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An object is defined twice (placed at different locations) in different modules.

L2096 Overlap variable <Name> not allocated

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Variables in segment _OVERLAP are only allocated together with the defining function. This message is issued if all the accesses to some overlap variable are removed, but the variable is defined and should be linked because smart linking is switched off.
The option -CAllocUnusedOverlap does change the default behavior so that such variables are allocated.
Note: If any not allocated variable is referenced, the linker does issue L2071.

See also

L2071: Fixup Error: Reference to non linked object (<objname>)
Option -CAllocUnusedOverlap
Overlapping Locals

L2097 Additional overlap variable <Name> allocated

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Variables in segment _OVERLAP are only allocated together with the defining function. If a function does not refer to one of its local variables, but this variable is still defined in the object file, this message is issued when allocating this variable. Such an variable is not used, and, even worse, its space is not overlapped with any other variable.
Additional overlap variables are only allocated when the option -CAllocUnusedOverlap is specified.

Tips

- Switch on SMART Linking when using overlapping.
- Modify the source that no such variables exist.
- Add a dependency of the defining function to this variable by using DEPENDENCY ADDUSE.

See also

Link Parameter File Command DEPENDENCY

Option -CAllocUnusedOverlap
Overlapping Locals

L2098 The label <labelname> cannot be resolved because of a recursion.

[DISABLE, INFORMATION, WARNING, ERROR]

Description

According to the input file depends the label on a recursive definition.
For example label is defined as label b plus some offset and label b is defined as label a plus some offset.

Tips

- Check the label definition.
- Rebuild the application, an object file might be corrupted.
- Move this error to a warning and check if there are other problems reported too.

L2103 Linking succeeded. Executable is written to <absfile>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Success message of the linker. In <absfile> the destination file of the link process (absolute file) is printed with full path.
Note that this message is disabled by default. It is only visible if it is explicitly enabled by a command line option.

See also

Command line option -WmsgSi.

L2104 Linking failed

[ERROR]

Description

Fail message of the linker. The specified destination file (absolute file) of the link process is deleted.

L2150 Illegal fixup offset (low bits) in <object> with offset <offset> to <object>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The linker cannot resolve the fixup/relocation for a relative 8bit fixup. For this relative fixup the offset has to be even, but it is not.

Tips

Contact support with your example. You may move this message to a warning so you could continue with linking, but code may not execute correctly at the location indicated in the message.

L2151 Fixup out of range (<low> <= <offset> <= <high>) in <object> with offset <offset> to <object>

[DISABLE, INFORMATION, WARNING, ERROR]

Description
The linker cannot resolve the fixup/relocation because the distance to the object is too far. A reason could be that you indicated e.g. that an object/segment is placed in a 8bit address area, but in the linker parameter file the object/segment is placed into a 16bit address area.

Tips
Check if declaration for the compiler/assembler matches your memory map provided to the linker (parameter file).
Contact support with your example. You may move this message to a warning so you could continue with linking, but code may not execute correctly at the location indicated in the message.

L2201 Listing file could not be opened

[DISABLE, INFORMATION, WARNING, <u>ERROR</u>]

Description
The listing output file of the link process could not be opened..

L2202 File for output %s could not be opened

[DISABLE, INFORMATION, WARNING, <u>ERROR</u>]

Description
The destination file of the link process (absolute file) could not be opened.

Tips
Check, if it is not opened for reading by any other process (Decoder, Debugger, HI-WAVE) or the if the destination folder or file is not marked as read only.

L2203 Listing of link process to <listfile>

[DISABLE, <u>INFORMATION</u>, WARNING, ERROR]

Description
The listing file is printed with full path, if its creation succeeded.

L2204 Segment <segment> is not allocated to any section

[<u>ERROR</u>]

Description
A special segment ("_OVERLAP") required by the linker is not allocated to any section.

Example
At the current linker version no other segment than "_OVERLAP" causes this message.

L2205 ROM libraries cannot have a function main (<main>)

[DISABLE, INFORMATION, WARNING, <u>ERROR</u>]

Description
A main function was defined in the absolute file linked to the application as ROM library. This is not allowed as default setting.

L2206 ROM libraries cannot have an INIT function (<init>)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An init function was defined in the absolute file linked to the application as ROM library. This function can cause a conflict when linking ton an application with an init function with the same name.

L2207 <main> not found

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The function main was not found in any of the linked modules.

L2208 No copydown created for initialized object "<Name>". Initialization data lost.

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The named object is allocated in RAM and it is defined with some initialization values. But because no copy down information is allocated, the initialization data is lost.

Example

```
int i= 19;
int j;
```

When linking the code above with no startup code, then the linker does issue this warning for i as its initialization value 19 is not used.

Tips

If you do not want this object to be initialized, change the source or ignore the warning.
If you want this object to be located in a ROM area, check the source. Do you specify option -CC with the compiler? Is the object constant or not?
If you want this object to be initialized at runtime, you need some startup code. You can use the one provided with the compiler or take it as example and adapt it to your needs.

L2251 Link parameter file <prmfile> not found

[ERROR]

Description

The link parameter file (extension .PRM), the source file of the linker, was not found. The specified source file does not exist or the search paths are not correctly set.

Tips

Check your “default.env” path settings. Link parameter files are searched in the current directory or in the list of paths specified with the environment variable ‘GENPATH’.

L2252 Illegal syntax in link parameter file: <syntaxerror>

[ERROR]

Description

A syntax error occurred in the link parameter file. The detailed error cause is printed in <syntaxerror>.

Examples (of <syntaxerror> messages)

"number too big"
"Comment not closed"
"hexadecimal number expected"
"unexpected end of file"

Tips

The cause of the errors reported here are syntactically and therefore easily detected are with the given source position info.

L2253 <definition> not present in link parameter file

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The definition of <definition> is not present in the link parameter file, but absolutely required by the link process.

Example

“NAMES definition is not present in the link parameter file”

Tips

Use a template link parameter file, for your target, where all these definitions are always present. Modify this file for your application. This way you avoid to write the same default settings for every application again and you will not forget definitions that have always to be present.

If START is specified as not present in the linker parameter file, then the reason for that could be that the application entry point of the application is not present, e.g. the ‘main’ routine is defined as ‘static’.

L2254 <definition> is multiply defined in link parameter file

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The definition of <definition> is allowed to be present only once but duplicated in the link parameter file.

Example

“PLACEMENT definition is duplicated in the link parameter file”

L2257 Both stacktop and stacksize defined

[ERROR]

Description

You can only define STACKTOP or STACK size, because a specification of one of them defines the settings of the other.

L2258 No stack definition allowed in ROM libraries

[DISABLE, INFORMATION, WARNING, ERROR]

Description

No stack definition allowed in ROM libraries.

L2259 No main function allowed in ROM libraries

[DISABLE, INFORMATION, WARNING, ERROR]

Description

No main function allowed in ROM libraries.

L2300 Segment <segmentname> not found in any objectfile

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A segment, declared in the link parameter file, was not found in any object file.

Tips:

Check this name in the linker parameter file and in the sources.
This message is issued to warn about possible spelling differences from a segment name in the source files and in the link parameter file.
If the link parameter file is shared between different projects and some of them do not have this segment, you can disable this message.

L2301 Segment <segmentname> must always be present

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Some segments are absolutely required by the linker. If they are not present an error is issued.

Example

"SSTACK" is such a segment if the linked file becomes an executable and not a ROM library.

L2303 Segment <seg1> has to be allocated into <seg2>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Segment <seg1> has to be allocated in <seg2>

Example
(For XA only) ROM_VAR has to be allocated in ROM section.

L2304 <segmentname> appears twice in the <deflist> definition list

[ERROR]

Description

The name <segmentname> appears twice in the definition list <deflist> of the link parameter file. <deflist> is either SECTIONS or PLACEMENTS.

Example

"MY_RAM appears twice in the SECTIONS definition list."

L2305 In link parameter file: The segment <segment> has the section type <type> (illegal)

[DISABLE, INFORMATION, WARNING, ERROR]

Description
The section type of segment <segment> is illegal.

L2306 Section <<seg1start>,<seg1end>> and Section <<seg2start>,<seg2end>> overlap

[DISABLE, INFORMATION, WARNING, ERROR]

Description
Segments are not allowed to overlap.

L2307 SSTACK cannot be allocated on more than one section

[ERROR]

Description
The stack has to be placed in one section.

L2308 Size of Stack (STACKSIZE = 0x<stacksize>) exceeds size of segment SSTACK (=0x<segmentsize>)

[ERROR]

Description
The STACKSIZE definition defines the size of the stack, that has to be placed in SSTACK. Therefore STACKSIZE is not allowed to exceed the size of SSTACK.

L2309 STACKTOP-command specifies 0x<stacktop> which is not in SSTACK (0x<stackstart>..0x<stackend>)

[ERROR]

Description
The STACKTOP definition defines the top address of the stack, that has to be placed in SSTACK. Therefore STACKTOP is not allowed to be outsize of the SSTACK segment.

L2310 The STACKTOP definition is incompatible with SSTACK being part of a list of segments

[ERROR]

Description

The STACKTOP definition in the link parameter file conflicts with the definition of the stack segment SSTACK in the link parameter file.

Tips
Change one of the definitions in the link parameter file.

L2311 STACKTOP or STACKSIZE missed

[DISABLE, INFORMATION, WARNING, ERROR]

Description
No STACKTOP or STACKSIZE declared, so no stack defined.

L2312 Stack not initialized

[DISABLE, INFORMATION, WARNING, ERROR]

Description
If the stack is defined, it has to be initialized.

L2313 All <segtype>_BASED segments must fit in a range of 64 kBytes

[ERROR]

Description
Based segments must be smaller that 64K.

L2314 A <segtype>_BASED segment must not have an address less than <address>

[ERROR]

Description
Only for the HC16.
Based segments must be smaller that 64K.

L2315 A <segtype>_BASED segment must not have an address bigger than <address>

[ERROR]

Description
Only for the HC16.
Based segments must be smaller that 64K.

L2316 All SHORT <segtype>_BASED segments must fit in a range of <range> Bytes (<startadr> - <endadr> > 256 Bytes)

[ERROR]

Description
Only for the HC16.
Based short segments must be smaller that 256 Bytes.

L2317 All non far segments have to be allocated on one single page

[ERROR]

Description
Only far segments can be allocated on multiple pages. All others have to allocated on a single page.

L2318 Cannot split _OVERLAP

[DISABLE, INFORMATION, WARNING, ERROR]

Description
The linker does not support to split the _OVERLAP segment into several areas in the HIWARE object file format. In the ELF object file format, this is supported.

Tips
Check if you can use the ELF object file format. Most of the build tools do support it as well as the HIWARE object file format.
Try to allocate the _OVERLAP first. Most other types of segments can be split into

several areas.

L2400 Memory model mismatch: <model1> (previous files) and model <model2> in module <objfile>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The memory model of an application to link has to be unique for all modules.
If this error is moved to a warning or less, then the application is generated. However, depending on the compilation units, the generated application might not work as different memory models do usually have different calling conventions. Also other problems might occur.
This message should only be moved by experienced users.

L2401 Target CPU mismatch: <cpu1> (previous files) and <cpu2> in module <objfile>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The memory model of an application to link has to be unique for all modules.
If this error is moved to a warning or less, then the application is generated. However, depending on the compilation units, the generated application might not work as different memory models do usually have different calling conventions. Also other problems might occur.
This message should only be moved by experienced users.

L2402 Incompatible flags or compiler options: <flags>

[ERROR]

Description

The flags set in an object file are incompatible with these of proceeding object files or with compiler options.

L2403 Incompatible flags or compiler options: <flags>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Same as L2402, but not an error, but a relocatable warning message.

L2404 Unknown processor: <processor> in module <modulename>

[ERROR]

Description

The target processor id is not recognized by the linker. This may be caused by the support of a target by the compiler that is not yet supported by the linker version used, or the object file is corrupt. Another cause may be an internal error in the Linker.

Tips

If recompiling leads to the same results, contact Metrowerks support for help.

L2405 Illegal address range in link parameter file. In the <model> memory model data must fit into one page

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Some memory models (SMALL, MEDIUM1) require the data segment to be allocated into one page.

L2406 More than one data page is used. Segment <segname> is in page 0

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Only for the HC16.

In the small memory model the data page must fit into page 0.

L2407 More than one data page is used in <memorymodel> memory model. The data page is defined by the placement of the stack

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Some memory models (SMALL, MEDIUM) require the data page to be defined in the same placement as the stack.

L2408 Illegal address range in link parameter file. In <memorymodel> memory model the code page must be page zero

[ERROR]

Description

Some memory models (SMALL) require the code page to be on page 0.

L2409 Multiple links are illegal: <object1>(module <module1>) links to <link1>(module <toModule1>) and to <link2>(module <toModule2>)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Inconsistency in the handling of unresolved imports. The importing object <object1> in the module <module1> found the definition of an external candidate object twice. (The first exporter is the object <link1> in the module <toModule1> and the second exporter is the object <link2> in the module <toModule2>.

L2410 Unresolved external <object> (imported from <module>)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An external imported from module <module> could not be found in any object file.

L2412 Dependency '<object>' description: '<description>’

[DISABLE, INFORMATION, WARNING, ERROR]

Description
This warning is issued if the linker cannot handle a part of a link parameter file command “DEPENDENCY”. Usually some of the named objects cannot be found.
The linker does not consider this <object> any more for the dependency information.

Tips
Check the spelling of all names. See in the mapfile how C++ name-mangled objects are called.

See also
Link parameter file command DEPENDENCY

L2413 Align STACKSIZE from <oldSize> to <newSize>

[DISABLE, INFORMATION, WARNING, ERROR]

Description
The stack size is aligned to a new value. The actual alignment needed depends on the target processor.

Tips
Specify an aligned size in the prm file, if your processor needs an aligned stack.

L2414 Stacksize not aligned. Is <oldsize>, expected to be aligned to <expectedsize>

[DISABLE, INFORMATION, WARNING, ERROR]

Description
The stack size is not aligned to an expected size. The actual alignment needed depends on the target processor.

Tips
Specify an aligned size in the prm file, if your processor needs an aligned stack.

L2415 Illegal dependency of '<object>'

[ERROR]

Description
This error is only generated for illegal object files. Check the producing tool.

L2416 Illegal file name '<Filename>'

[ERROR]

Description
The specified filename is was not correctly terminated. This error may happen if a filename is specified with a single double quotes.

Example
LINK “a.abs
...

Tips
Terminate the file name with a second double quote.

L2417 Object <objname> refers to non existing segment number

<segnumber>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The specified object refers to a segment number which is not defined in the segment table of the object file. This error only occurs for illegal, corrupted object files.

Tips

Delete the object file, and rebuild it. If the error occurs again, contact the vendor of the object file producing tool.

If this error is ignored, the default ROM/RAM segment is assumed.

L2418 Object <objname> allocated in segment <segname> is not allocated according to the segment attribute <attrname>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The linker has found an object, allocated in a segment with a special segment attribute, which was not allocated according to this attribute.

This warning occurs when the source code attributes do not correspond to the memory area specified for the segment.

Example

Note: This example generates the warning only for target compilers supporting the SHORT segment modifier.

C source file (test.c):

```
#pragma DATA_SEG SHORT SHORT_SEG
int i;
void main(void) {
  i=1;
}
```

prm file (test.prm)

```
LINK test.abs
NAMES test.o END
SECTIONS
  MY_RAM = NO_INIT     0x180 TO  0x1ff;
  MY_ROM = READ_ONLY  0x1000 TO 0x1fff;
PLACEMENT
  DEFAULT_ROM INTO MY_ROM;
  DATA_SEG, _OVERLAP, DEFAULT_RAM INTO MY_RAM;
END
INIT main
```

Tips

Check your sources and your link parameter file if the handle the named object and segment correctly.

Messages Independent of the Object File Format

L1: Unknown message occurred

[FATAL]

Description

The linker tried to emit a message which was not defined.This is a internal error which should not occur. Please report any occurrences to you distributor.

Tips

Try to find out the and avoid the reason for the unknown message.

L2: Message overflow, skipping <kind> messages

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The linker did show the number of messages of the specific kind as controlled with the options -WmsgNi, -WmsgNw and -WmsgNe. Further options of this kind are not displayed.

Tips

Use the options -WmsgNi, -WmsgNw and -WmsgNe to change the number of messages

L50: Input file '<file>' not found

[FATAL]

Description

The Application was not able to find a file needed for processing.

Tips

Check if the file really exits. Check if you are using a file name containing spaces (in this case you have to quote it).

L51: Cannot open statistic log file <file>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

It was not possible to open a statistic output file, therefore no statistics are generated. Note: Not all tools do support statistic log files. Even if a tool does not support it, the message does still exist, but is never issued in this case, of course.

L52: Error in command line <cmd>

[FATAL]

Description

In case there is an error while processing the command line, this message is issued.

L64: Line Continuation occurred in <FileName>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

In any environment file, the character '\' at the end of a line is taken as line continu-

ation. This line and the next one are handles as one line only. Because the path separation character of MS-DOS is also '\', paths are often incorrectly written ending with '\'. Instead use a '.' after the last '\' to not finish a line with '\' unless you really want a line continuation.

Example:

Current Default.env:

```
...
LIBPATH=c:\metrowerks\lib\
OBJPATH=c:\metrowerks\work
...
```

Is taken by the compiler identical as

```
...
LIBPATH=c:\metrowerks\libOBJPATH=c:\metrowerks\work
...
```

Tips

To fix it, append a '.' behind the '\'

```
...
LIBPATH=c:\metrowerks\lib\.
OBJPATH=c:\metrowerks\work
...
```

Note:

Because this information occurs during the initialization phase of the compiler, the 'C' might not occur in the error message. So it might occur as "64: Line Continuation occurred in <FileName>".

L65: Environment macro expansion message '<description>' for <variablename>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

During a environment variable macro substitution an problem did occur. Possible causes are that the named macro did not exist or some length limitation was reached. Also recursive macros may cause this message.

Example

Current variables:

```
...
LIBPATH=${LIBPATH}
...
```

Tips

Check the definition of the environment variable.

L66: Search path <Name> does not exist

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The tool did look for a file which was not found. During the failed search for the file,

a non existing path was encountered.

Tips

Check the spelling of your paths. Update the paths when moving a project. Use relative paths.

L4000 Could not open object file (<objFile>) in NAMES list

[ERROR]

Description

The linker could not open any object file in the NAMES list. This message prints out the name of the last file in the names list found (<objFile>).

Tips

Check your “default.env” path settings. Object files are searched in the current directory or in the list of paths specified with the environment variables ‘OBJPATH’ and ‘GENPATH’.

L4001 Link parameter file <PRMFile> not found

[ERROR]

Description

The specified source file does not exist or the search paths are not correctly set.

Tips

Check your “default.env” path settings. Link parameter files are searched in the current directory or in the list of paths specified with the environment variable ‘GENPATH’.

L4002 NAMES section was not found in link parameter file <PRMFile>

[ERROR]

Description

The new HIWARE Linker detects the object file format to link by scanning the NAMES section for the first file that it can open to evaluate the file format. If the NAMES section was not found in the link parameter file, this message is issued.

Tips

Look if the file passed to the linker is really a link parameter file.

L4003 Linking <PRMFile> as HIWARE format link parameter file

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The new HIWARE Linker detects the object file format to link by scanning the NAMES section for the first file that it can open to evaluate the file format. If the first file in the NAMES section, that can be opened by the linker is a HIWARE object file, this message is issued and the HIWARE object file format linker, a subprocess of the HIWARE Linker is started.
Note that this message is disabled by default. It is only issued if the message is explicitly enabled on the command line.

See also
Command line option -WmsgSi.

L4004 Linking <PRMFile> as ELF/DWARF format link parameter file

[DISABLE, INFORMATION, WARNING, ERROR]

Description
The new HIWARE Linker detects the object file format to link by scanning the NAMES section for the first file that it can open to evaluate the file format. If the first file in the NAMES section, that can be opened by the linker is a ELF/DWARF object file, this message is issued and the ELF/DWARF object file format linker, a subprocess of the HIWARE Linker is started.
Note that this message is disabled by default. It is only issued if the message is explicitly enabled on the command line.

See also
Command line option -WmsgSi.

L4005 Illegal file format of object file (<objFile>)

[ERROR]

Description
There is no object file in the NAMES list with a known file format or a object file specified with option -add has a unknown file format.

Tips
Check your “default.env” path settings. Object files are searched in the current directory or in the list of paths specified with the environment variables ‘OBJPATH’ and ‘GENPATH’. It may be that you have files of another development environment in your directories.

L4006 Failed to create temporary file

[ERROR]

Description
The linker creates a temporary file for the prescan of the link parameter file in the current directory. If this fails, the Linker can’t continue.

Tips
Enable the read access to files for the Linker in the current directory.

L4007 Include file nesting to deep in link parameter file

[DISABLE, INFORMATION, WARNING, ERROR]

Description
Only an include file nesting of maximum depth 6 is allowed.

L4008 Include file <includefile> not found

[DISABLE, INFORMATION, WARNING, ERROR]

Description
The include file <includefile> was not found.

L4009 Command <Command> overwritten by option <Option>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

This message is generated, when a command from the PRM file is overwritten by a command line option.

<command name>: name of the command, which is overwritten by a linker option
<option name>: linker options, which overwrites the linker command.

In this case the command line option is stronger than the command specified in the PRM file. The commands, which may be overwritten by a command line option, are:

- LINK, which may be overwritten by the option –O (definition of the output file name).
- MAPFILE, which may be overwritten by the option –M (enable generation of the MAP file).
- INIT, which may be overwritten by the option –E (definition of the application entry point).

When the LINK command is detected in the PRM file and the option –O is specified on the command line, following message is generated:

`'Command LINK overwritten by option -O'`

Tips

Remove either the command in the PRM file or the option on the command line.

L4010 Burner file creation error '<Description>'

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The built-in burner was not able to generate an output file because of the given reason.

Tips

The application (*.abs) is still generated correctly. You might use the external burner to produce the file.

L4011 Failed to generate distribution file because of <reason>

[DISABLE, INFORMATION, WARNING,ERROR]

Description

Failed to generate a distribution file because of the given reason <reason>.

See also

Option -Dist
Section Automatic Distribution of Variables

L4012 Failed to generate distribution file because of distribution segment <segment> not found or not alone in placement

[DISABLE, INFORMATION, WARNING, ERROR]

Description

This message is generated, when the distribution segment <segment> doesn't exist in the placement of the PRM file or if it doesn't stay alone in the placement.

Example

If DISTRIBUTE is the distribution segment <segment>, it has to stay ALONE in the placement.

Then it should look as follows:

```
PLACEMENT
  DISTRIBUTE DISTRIBUTE_INTO MY_ROM0, MY_ROM1;
```

See also

Option -Dist

Section Automatic Distribution of Variables

L4013 Function <function> is not in the distribution segment

[DISABLE, INFORMATION, WARNING, ERROR]

Description

If a function inside of the distribution segment is called from a outside one (the one mentioned in the message), it has to have a far calling convention. This has a negative influence of the optimization. This message is generated to have an overview from which outside functions an incoming call exist.

Tips

If it‘s possible, insert this functions in the distribution segment.

See also

Option -Dist

Section Automatic Distribution of Variables

L4014 The processor <processor> is not supported by the linker optimizer

[DISABLE, INFORMATION, WARNING, ERROR]

Description

This message is generated, when the processor is not supported by the linker optimizer.

Tips

If your target CPU has to be supported with this optimization, please check with support if this could be done with a new release.

L4015 Section <section> has no IBCC_NEAR or IBCC_FAR flag

[DISABLE, INFORMATION, WARNING, ERROR]

Description

This message is generated, when a section <section> which is in the distribution segment doesn‘t have an IBCC_NEAR (inter bank calling convention near) or an IBCC_FAR (inter bank calling convention far) Flag.

Example

Each section in the PLACEMENT list used for the distribution (DISTRIBUTE_INTO) has either to have the IBCC_NEAR or the IBCC_FAR flag.

```
SECTIONS
  MY_ROM0   = READ_ONLY  IBCC_NEAR 0x005000 TO 0x00504F;
```

```
  MY_ROM1  = READ_ONLY  IBCC_FAR  0x018000 TO 0x018050;
  MY_ROM2  = READ_ONLY  IBCC_FAR  0x028000 TO 0x0280F0;
END
PLACEMENT
  DISTRIBUTE DISTRIBUTE_INTO MY_ROM0, MY_ROM1, MY_ROM2;
END
```

See also

Option -Dist

Section Automatic Distribution of Variables

L4016 No section in the segment <segment> has an IBCC_NEAR flag

[DISABLE, INFORMATION, WARNING, ERROR]

Description

This message is generated, when no section which is in the distribution segment <segment> has an IBCC_NEAR Flag (inter bank calling convention).

Tips

Check if you really don't want to have distributed functions in a 'near' section. Placing functions in a 'near' section may increase performance and could improve code density.

See also

Option -Dist

Section Automatic Distribution of Variables

L4017 Failed to generate distribution file because there are no functions in the distribution segment <segment>

[ERROR]

Description

This message is generated, when no functions are found in the code segment <Segment>. <Segment> is the name of the distribution segment, which contains the functions for the optimized distribution. For the Linker optimizer it is necessary to specify in the source files a command like: #pragma CODE_SEG <Segment>. All functions which follow this command are automatically distributed into this Segment.

Tips

Check if your compiler supports the "#pragma CODE_SEG".
Recompile the source files without to include the distribution file.
Check if in the sources the command #pragma CODE_SEG <Segment> really exists.

See also

Option -Dist

Section Automatic Distribution of Variables

L4018 The sections in the distribution segment have not enough memory for all functions

[ERROR]

Description

This message is generated, when the functions which were distributed into the special distribution segment have not enough space into the sections of it.

Tips

Add more pages to the distribution segment or increase the size of this pages.
If this not help decrease the amount of functions in the distribution segment.

See also

Option -Dist
Section Automatic Distribution of Variables

L4019 Function <function name> has a near flag and can not be distributed

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The linker optimizer doesn‘t support functions which are assigned in the source code with a near flag.

Example

MyFunction is distributed in the distribution segment “DISTRIBUTE” and has a near flag:

```
#pragma CODE_SEG DISTRIBUTE
void near MyFunction(void) {}
```

Tips

Avoid to use the near Flag (e.g.: void near MyFunction(void) {}) for functions which have to be distributed in the distribution segment.

See also

Option -Dist
Section Automatic Distribution of Variables

L4020 Not enough memory in the non banked sections of the distribution segment <segment>

[ERROR]

Description

While optimizing functions out of the distribution segment, the linker has not found enough memory in the non banked sections of the distribution segment. Only functions which have a near flag can be placed in a non banked section.

Tips

All near functions must have enough space in the near sections (sections with the IBCC_NEAR flag) of the distribution segment. If possible increase the size of the non banked sections of the distribution segment, otherwise remove some near functions from it.

See also

Option -Dist

Section Automatic Distribution of Variables

L4021 Incompatible derivative: <Deriv0> in previous files and <Deriv1> in current file

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The two mentioned object files were compiled or assembled for different, incompatible derivatives of the same CPU family.
Depending on which features of the two derivatives were used, the generated executable might not work for the one or the other derivative (or even for none of them).

Tips

Recompile your sources, and use a common setting for all source files.
Some compilers/assembler do provide a generic mode, which does not use the specific features not available in all derivatives.

L4022 HexFile not found: <Filename>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The hexfile <Filename> to link with (specified with the HEXFILE command in the link parameter file) was not found. The specified hex file does not exist or the search paths are not correctly set.

Tips

Check your “default.env” path settings. Hex files are searched in the current directory or in the list of paths specified with the environment variable ‘GENPATH’.

L4023 Hexfile error '<Description>' in file '<Filename>'

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The linker did find some problems with the hexfile <Filename>.
Possible problems are a bad checksum, a bad length, a too large length (>256) or an otherwise corrupted file.

Tips

Check, if the file specified is really a hex file. If yes, create it and try again.
The decoder can be used to check Motorola S Record files.

L4024 No information available for segment '<name>'.

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A undefined symbol started with one of the linker defined prefixes "__SEG_START_", "__SEG_END_" or "__SEG_SIZE_"
but the name of the following segment was not known. Therefore the linker does not know to which address this symbol should evaluate. To handle this case, the linker does issue this message and the linker is using the address 0 as address.
However, the linking does not fail.

Example

```
extern char __SEG_START_UNKNOWN_SEGMENT[];
```

Tips

Check the spelling. Do explicitly name the segment in the link parameter file.

L4025 This limited version allows only <num> <limitKind>

[ERROR]

Description

Depending on your license configuration, the linker may e.g. limited only to allow up to 4K C++ code. The limitation size you will see from the <num> field and the limitation kine (e.g. C++ code) you can see from the <limitKind> field. The limitations are also shown in the about box.

Tips

Check if you are using a correct license configuration. Contact your vendor for a unlimited license or upgrade.

Index

Symbols

#pragma CODE_SEG 354
.abs 15, 66
.copy 216, 226, 287
.data 216, 217
.hidefaults 39, 40, 55, 63
.ini 22
.init 217
.map 66, 239
.overlap 139, 217
.prm 65
.rodata 216
.rodata1 216
.s1 66
.s2 66
.s3 66
.stack 216, 217
.startData 216, 217, 226
.sx 66
.text 216, 217
__DEFAULT_SEG_CC__ 154
__INTERSEG_CC__ 153
__INTRAPAGE__ 152
__NON_INTERSEG_CC__ 153
__SEG_END_ 149
__SEG_END_DEF 150
__SEG_END_REF 150
__SEG_SIZE_ 149
__SEG_SIZE_DEF 150
__SEG_SIZE_REF 150
__SEG_START_ 149
__SEG_START_DEF 150
__SEG_START_REF 150
__SEG_START_SSTACK 149
_OVERLAP 139, 221
_PRESTART 221

A

About Box 33
Absolute File 15, 65, 187, 193
ABSPATH 53, 65, 66, 168
-Add 72
-Alloc 73
Application
    Startup (also see Startup) 225
-AsROMLib 75
Assembly
    Application 160
    LINK_INFO 164
    Prm File 160
    Smart Linking 161
AUTOLOAD 171
Automatic Distribution 151

B

-B 76

C

-CAllocUnusedOverlap 76
CHECKKEYS 176
CHECKSUM 172
Checksum Computation 156
-Ci 77
-Cocc 78
CODE 123
CodeWarrior 89, 168
color 96, 97, 98
Command
    AUTOLOAD 171
    CHECKKEYS 176
    CHECKSUM 172
    DATA 176
    DEPENDENCY 177
    ENTRIES 130, 131, 132, 133, 182, 241
    HAS_BANKED_DATA 184
    HEXFILE 185
    INIT 186, 241
    LINK 89, 168, 187, 241
    MAIN 188, 241
    MAPFILE 86, 189
    NAMES 133, 134, 168, 192, 242
    OVERLAP_GROUP 194
    PLACEMENT 124, 168, 196, 217, 221
    PRESTART 198
    SECTIONS 121, 199
    SEGMENTS 115, 168, 202

STACKSIZE 208
STACKTOP 210
START 211
VECTOR 129, 212
Common Code 207
COPY 220, 230, 287
COPYRIGHT 54, 59, 64
-CRam 79
Current Directory 55
CurrentCommandLine 47

D

DATA 176
Default Directory 41
DEFAULT.ENV 39, 40, 55, 63
DEFAULT_RAM 220, 221
DEFAULT_ROM 220, 221
DEFAULTDIR 40, 41, 54
DefaultDir 41
DEPENDENCY 177
Dependency 66
-Dist 79
-DistFile 80
-DistInfo 80
-DistOpti 81
DISTRIBUTE_INTO 155
Distribution Segment 153
-DistSeg 81

E

-E 82
Editor 46
Editor_Exe 44, 46
Editor_Name 43, 46
Editor_Opts 44, 46
EditorCommandLine 49
EditorDDEClientName 49
EditorDDEServiceName 50
EditorDDETopicName 49
EditorType 49
EDOUT 68
ENTRIES 130, 131, 132, 133, 182, 241
-Env 39, 82
ENVIRONMENT 55
Environment Variable 39, 52
ABSPATH 53, 65, 66, 168, 187
COPYRIGHT 54, 59, 64
DEFAULTDIR 40, 41, 54
ENVIRONMENT 55
ENVIRONMENT 39
ERRORFILE 56, 67
GENPATH 58, 60, 65, 168, 193
HIENVIRONMENT 55
INCLUDETIME 54, 59, 64
LINKOPTIONS 71
LINKPTIONS 60, 71
OBJPATH 60, 168, 193
RESETVECTOR 61
SRECORD 61, 66
TEXTPATH 62, 66, 168, 187
TMP 63
USERNAME 54, 59, 63
Error File 67
Error Listing 67
ERRORFILE 56, 67
Explorer 40

F

-F 83
File
Absolute 15, 65, 187, 193
Error 67
Library 193
Map 66, 187, 190, 239
Motorola S 66
Object 65, 193
Parameter 65
Parameter (Linker) 165
File Manager 40

G

GENPATH 58, 168, 193
Group 41

H

-H 84
HAS_BANKED_DATA 184
HEXFILE 185
HIENVIRONMENT 55

I

IBCC_FAR 155, 353
IBCC_NEAR 155, 353, 355
INCLUDETIME 54, 59, 64
INIT 186, 241

L

-L 84
Library File 193
-Lic 85
-Lica 85
LINK 89, 168, 187, 241
Linker
  Configuration 22
  Input File 65
  Menu 29
  Menu Bar 21
  Message Settings Dialog Box 31
  Messages 31
  Options 30
  Output Files 65
  Status Bar 21
  Tool Bar 20
LINKOPTIONS 45, 60

M

-M 86
MAIN 188, 241
map 187
Map File 66, 187, 190, 239
  COPYDOWN 240
  DEPENDENCY TREE 240
  FILE 239
  OBJECT ALLOCATION 239
  OBJECT DEPENDENCY 239
  SEGMENT ALLOCATION 239
  STARTUP 239
  STATISTICS 240
  TARGET 239
  UNUSED OBJECTS 240
MAPFILE 86, 189
MCUTOOLS.INI 24, 40, 55
Message
  ERROR 257
  FATAL 257
  WARNING 257
Motorola S File 66

N

-N 86
NAMES 133, 134, 168, 192, 242
NO_INIT 118, 123, 199, 203
-NoBeep 87
-NoEnv 87

O

-O 89
Object File 65, 193
OBJPATH 60, 168, 193
-OCopy 88
Option Settings Dialog 30
Options 41, 49
OVERLAP_GROUP 194
OVERLAYS 135

P

PAGED 118, 123, 135, 200, 203
Parameter
  File (Linker) 165
Parameter File 65
Path 41
Path List 51
PLACEMENT 124, 168, 196, 217, 221, 353
PRESTART 198
Prm file controlled Checksum Computation 157
-Prod 46, 89
Program Startup (also see Startup) 225
Project Directory 40
project.ini 46

Q

Qualifier 115, 117, 121, 123, 199, 202
  CODE 123
  NO_INIT 118, 123, 199, 203
  PAGED 118, 123, 200, 203
  READ_ONLY 117, 123, 199, 203
  READ_WRITE 117, 123, 199, 203

R

READ_ONLY 117, 123, 199, 203
READ_WRITE 117, 123, 199, 203

REALLOC_OBJ 153
RecentCommandLineX 47
RGB 96, 97, 98
ROM Library 241
ROM library 75, 187, 189, 193, 226, 231
ROM_LIB 187, 241
ROM_VAR 220

S

-S 90
SaveAppearance 42
SaveEditor 42
SaveOnExit 42
SaveOptions 42
Section 215, 219
- .copy 216, 226
- .data 216, 217
- .init 217
- .overlap 217
- .rodata 216
- .rodata1 216
- .stack 216, 217
- .startData 216, 217, 226
- .text 216, 217
- Pre-defined 216
- Qualifier 121, 123
- rodata 216

SECTIONS 121, 199
Segment 215, 219
- _OVERLAP 221
- _PRESTART 221
- Alignment 115, 119, 202, 205
- COPY 220, 230
- DEAFULT_RAM 220
- DEFAULT_RAM 221
- DEFAULT_ROM 220, 221
- Fill Pattern 115, 120, 206
- fill pattern 202
- Optimizing Constants 207
- Pre-defined 220
- Qualifier 115, 117, 199, 202
- ROM_VAR 220
- SSTACK 220, 221
- STARTUP 220, 221, 230
- STRINGS 220

SEGMENTS 115, 168, 202
-SFixups 91
ShowTipOfDay 43
Smart Linking 16, 130, 131
SSTACK 220, 221
STACK 209
STACKSIZE 208
STACKTOP 210
START 211
STARTUP 220, 221, 230
Startup
- Application 225

startup 46
Startup Function 230, 232
- User Defined 230, 232

Startup Structure 225, 230
- finiBodies 228
- flags 226, 231
- initBodies 228
- libInits 228, 231
- main 227, 231
- mInits 232
- nofFiniBodies 228
- nofInitBodies 228
- nofLibInits 228
- nofZeroOuts 227, 231
- pZeroOut 227, 231
- stackOffset 227, 231
- toCopyDownBeg 227, 231
- User Defined 228

Startup.TXT 225
-StatF 91
StatusbarEnabled 47
STRINGS 220

T

TEXTPATH 62, 66, 168
Tip of the Day 17
TipFilePos 42
TipTimeStamp 43
TMP 63
ToolbarEnabled 48

U

UNIX 40
USERNAME 54, 59, 63

V

-V 92
VECTOR 129, 212
Vector 16
-View 92

W

-W1 93
-W2 94
-WErrFile 95
WindowFont 48
WindowPos 48
Windows 40
WinEdit 40
-Wmsg8x3 95
-WmsgCE 96
-WmsgCF 97
-WmsgCI 97
-WmsgCU 98
-WmsgCW 98
-WmsgFb 36, 99
-WmsgFi 36, 100
-WmsgFob 101
-WmsgFoi 102
-WmsgFonf 104
-WmsgFonp 106
-WmsgNe 107
-WmsgNi 108
-WmsgNu 108
-WmsgNw 109
-WmsgSd 110
-WmsgSe 110
-WmsgSi 111
-WmsgSw 111
-WOutFile 112
-WStdout 112