# Table of Contents

## Introduction

- CodeWarrior IDE Utilities ................................................. 29
- SmartLinker ........................................................................ 29
- Burner Utility ................................................................. 29
- Libmaker ............................................................................ 29
- Decoder ............................................................................... 29
- Maker: The Make Tool ..................................................... 29
- Starting a CodeWarrior Utility ........................................... 30

## SmartLinker

- Manual Notation .............................................................. 31
- Purpose of a Linker ......................................................... 32
  - Product Features ......................................................... 32
  - Section Contents ....................................................... 32
- Starting the SmartLinker Utility ........................................ 33

## SmartLinker User Interface

- SmartLinker Main Window ............................................... 35
  - Window Title ............................................................. 36
  - Content Area ........................................................... 36
  - Main Window Tool Bar .............................................. 37
  - Main Window Status Bar .......................................... 38
  - Main Window Menu Bar ............................................ 38
  - SmartLinker Configuration Window ............................ 41
  - Options Settings Window ......................................... 50
  - Message Settings Window ......................................... 52
  - About Dialog Box .................................................... 55
# Table of Contents

- Retrieving Information About an Error Message ........................................ 56
- Specifying the Input File ........................................................................... 56
- Using the Command Line in the Tool Bar to Link .................................... 56
- Message/Error Feedback ........................................................................... 57

## 2 Environment Variables

### The Current Directory ............................................................................ 60
### Global Initialization File (MCUTOOLS.INI - PC Only) ............................ 61
  - [Installation] Section ............................................................................. 61
  - Path ........................................................................................................ 61
  - Group ..................................................................................................... 62
  - [Options] Section .................................................................................. 62
  - DefaultDir ............................................................................................. 62
  - [LINKER] Section .................................................................................. 63
  - SaveOnExit ............................................................................................ 63
  - SaveAppearance ..................................................................................... 63
  - SaveEditor ............................................................................................. 63
  - SaveOptions ........................................................................................... 64
  - RecentProject0, RecentProject1 ............................................................. 64
  - TipFilePos ............................................................................................. 64
  - ShowTipOfDay ....................................................................................... 65
  - TipTimeStamp ....................................................................................... 65
  - [Editor] Section .................................................................................... 66
  - Editor_Name .......................................................................................... 66
  - Editor_Exe ............................................................................................ 66
  - Editor_Opts ......................................................................................... 66
  - MCUTOOLS.INI Example ................................................................. 68
### Local Configuration File (Usually project.ini) ........................................ 68
  - [Editor] Section .................................................................................... 69
  - Editor_Name .......................................................................................... 69
  - Editor_Exe ............................................................................................ 70
  - Editor_Opts ........................................................................................... 70
  - [LINKER] Section ................................................................................ 71
  - RecentCommandLineX, X=Integer ......................................................... 71
  - CurrentCommandLine ................................................................. 71
# Table of Contents

- StatusBarEnabled .................................................. 72
- ToolbarEnabled ......................................................... 72
- WindowPos ............................................................... 73
- WindowFont ............................................................. 73
- Options ................................................................. 74
- EditorType ............................................................. 74
- EditorCommandLine ..................................................... 75
- EditorDDEClientName .................................................. 75
- EditorDDETopicName ................................................... 76
- EditorDDEServiceName ................................................ 76
- Configuration File Example ........................................... 77
- Paths ................................................................. 77
- Line Continuation ........................................................ 78
- Environment Variable Details ........................................ 79
  - ABSPATH: Absolute Path ............................................. 80
  - COPYRIGHT: Copyright Entry in Absolute File ................. 81
  - DEFAULTDIR: Default Current Directory ....................... 82
  - ENVIRONMENT: Environment File Specification ............... 83
  - ERRORFILE: Error File Name Specification .................... 84
  - GENPATH: Define Paths to Search for Input Files .............. 85
  - INCLUDETIME: Creation Time in Object File .................. 86
  - LINKOPTIONS: Default SmartLinker Options ................... 87
  - OBJPATH: Object File Path ........................................ 87
  - RESETVECTOR: Reset Vector Location ........................... 88
  - SRECORD: S Record File Format .................................. 89
  - TEXTPATH: Text Path ............................................... 90
  - TMP: Temporary Directory ....................................... 91
  - USERNAME: User Name in Object File ............................ 92

## 3 SmartLinker Files

- Input Files ............................................................ 93
- Parameter File ......................................................... 93
- Object File ............................................................ 93
- Output Files ........................................................... 94
- Absolute Files ......................................................... 94
Table of Contents

S Record Files ......................................................... 94
Map Files .......................................................... 94
Error Listing File ...................................................... 96

4 SmartLinker Options .............................................. 99
SmartLinker Option Details ............................................ 100
-Add: Additional Object/Library File ............................... 100
-Alloc: Allocation Over Segment Boundaries (ELF) ............... 101
-AsROMLib: Link as ROM Library .................................. 103
-B: Generate S-Record file .......................................... 103
-CAlocUnusedOverlap: Allocate Not Referenced Overlap Variables
  (Freescale) ............................................................. 104
-Ci: Link Case Insensitive ........................................... 105
-Cocc: Optimize Common Code (ELF) ............................... 106
-CRam: Allocate Non-specified Constant Segments in RAM (ELF) . 106
-Dist: Enable Distribution Optimization (ELF) ..................... 107
-DistFile: Specify Distribution File Name (ELF) .................. 107
-DistInfo: Generate Distribution Information File (ELF) ........ 108
-DistOpti: Choose Optimizing Method (ELF) ...................... 109
-DistSeg: Specify Distribution Segment Name (ELF) ............. 109
-E: Define Application Entry Point (ELF) ........................ 110
-Env: Set Environment Variable ..................................... 111
-FA, -FE, -FH -F6: Object File Format ............................. 111
-H: Prints the List of All Available Options ....................... 112
-L: Add a Path to Search Path (ELF) ............................... 113
-Lic: Print License Information .................................... 113
-LicA: License Information About Every Feature in Directory .... 114
-LicBorrow: Borrow License Feature ................................ 114
-LicWait: Wait Until Floating License Is Available from Floating
  License Server ......................................................... 115
-M: Generate Map File ................................................. 116
-N: Display Notify Box ................................................ 117
-NoBeep: No Beep in Case of an Error ............................. 117
-NoEnv: Do Not Use Environment ................................... 118
-O: Define Absolute File Name ...................................... 118
Table of Contents

-OCopy: Optimize Copy Down (ELF) ........................................ 119
-Prod: Specify Project File at Startup (PC) .......................... 120
-S: Do Not Generate DWARF Information (ELF) ......................... 121
-SFixups: Creating Fixups (ELF) ........................................ 121
-StatF: Specify Name of Statistic File ................................. 122
-V: Prints SmartLinker Version ........................................... 122
-View: Application Standard Occurrence (PC) ......................... 123
-W1: No Information Messages ............................................ 124
-W2: No Information and Warning Messages ......................... 124
-WErrFile: Create “err.log” Error File ................................ 125
-Wmsg8x3: Cut File Names in Microsoft Format to 8.3 (PC) ....... 126
-WmsgCE: RGB Color for Error Messages ............................... 127
-WmsgCF: RGB Color for Fatal Messages ............................... 127
-WmsgCI: RGB Color for Information Messages ...................... 128
-WmsgCU: RGB Color for User Messages ............................... 129
-WmsgCW: RGB Color for Warning Messages ......................... 129
-WmsgFb (-WmsgFbv, -WmsgFbm): Set Message File Format for Batch Mode ..................................................... 130
-WmsgFi (-WmsgFiv, -WmsgFim): Set Message File Format for Interactive Mode ..................................................... 132
-WmsgFob: Message Format for Batch Mode ............................ 133
-WmsgFoi: Message Format for Interactive Mode ...................... 135
-WmsgFonf: Message Format for no File Information ................. 137
-WmsgNe: Number of Error Messages .................................. 140
-WmsgNi: Number of Information Messages ........................... 140
-WmsgNu: Disable User Messages ........................................ 141
-WmsgNw: Number of Warning Messages ............................... 142
-WmsgSe: Setting a Message to Disable .............................. 143
-WmsgSi: Setting a Message to Information ......................... 144
-WmsgSw: Setting a Message to Warning .............................. 145
-WOutFile: Create Error Listing File .................................... 146
-WStdout: Write to Standard Output .................................... 147
# 5 Linking Issues

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Allocation</td>
<td>149</td>
</tr>
<tr>
<td>The SEGMENTS Block (ELF)</td>
<td>149</td>
</tr>
<tr>
<td>The SECTIONS Block (Freescale (Hiware) + ELF)</td>
<td>155</td>
</tr>
<tr>
<td>PLACEMENT Block</td>
<td>158</td>
</tr>
<tr>
<td>Initializing Vector Table</td>
<td>163</td>
</tr>
<tr>
<td>VECTOR Command</td>
<td>163</td>
</tr>
<tr>
<td>Smart Linking (ELF)</td>
<td>163</td>
</tr>
<tr>
<td>Mandatory Linking of an Object</td>
<td>164</td>
</tr>
<tr>
<td>Mandatory Linking of all Objects Defined in Object File</td>
<td>164</td>
</tr>
<tr>
<td>Switching OFF Smart Linking for the Application</td>
<td>165</td>
</tr>
<tr>
<td>Smart Linking (Freescale + ELF)</td>
<td>165</td>
</tr>
<tr>
<td>Mandatory Linking from an Object</td>
<td>165</td>
</tr>
<tr>
<td>Mandatory Linking from all Objects defined in a File</td>
<td>166</td>
</tr>
<tr>
<td>Binary Files Building an Application (ELF)</td>
<td>167</td>
</tr>
<tr>
<td>NAMES Block</td>
<td>167</td>
</tr>
<tr>
<td>ENTRIES Block</td>
<td>167</td>
</tr>
<tr>
<td>Binary Files Building an Application (Freescale former Hiware)</td>
<td>168</td>
</tr>
<tr>
<td>NAMES Block</td>
<td>168</td>
</tr>
<tr>
<td>Allocating Variables in “OVERLAYS”</td>
<td>169</td>
</tr>
<tr>
<td>Overlapping Locals</td>
<td>170</td>
</tr>
<tr>
<td>Algorithm</td>
<td>171</td>
</tr>
<tr>
<td>Name Mangling for Overlapping Locals</td>
<td>173</td>
</tr>
<tr>
<td>Name Mangling in ELF Object File Format</td>
<td>173</td>
</tr>
<tr>
<td>Defining a Function with Overlapping Parameters in Assembler</td>
<td>174</td>
</tr>
<tr>
<td>DEPENDENCY TREE Section in Map File</td>
<td>179</td>
</tr>
<tr>
<td>Optimizing the Overlap Size</td>
<td>180</td>
</tr>
<tr>
<td>Recursion Checks</td>
<td>180</td>
</tr>
<tr>
<td>Linker Defined Objects</td>
<td>182</td>
</tr>
<tr>
<td>Automatic Distribution of Paged Functions</td>
<td>184</td>
</tr>
<tr>
<td>Limitations</td>
<td>188</td>
</tr>
<tr>
<td>Checksum Computation</td>
<td>189</td>
</tr>
<tr>
<td>prm File-Controlled Checksum Computation</td>
<td>190</td>
</tr>
<tr>
<td>Automatic Linker Controlled Checksum Computation</td>
<td>190</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Partial Fields</td>
<td>192</td>
</tr>
<tr>
<td>Runtime Support</td>
<td>192</td>
</tr>
<tr>
<td>Linking an Assembly Application</td>
<td>192</td>
</tr>
<tr>
<td>prm File</td>
<td>192</td>
</tr>
<tr>
<td>Warnings</td>
<td>193</td>
</tr>
<tr>
<td>Smart Linking</td>
<td>193</td>
</tr>
<tr>
<td>LINK_INFO (ELF)</td>
<td>196</td>
</tr>
<tr>
<td>6 SmartLinker Parameter File</td>
<td>197</td>
</tr>
<tr>
<td>Syntax of the Parameter File</td>
<td>197</td>
</tr>
<tr>
<td>Mandatory SmartLinker Commands</td>
<td>199</td>
</tr>
<tr>
<td>The INCLUDE Directive</td>
<td>200</td>
</tr>
<tr>
<td>7 SmartLinker Commands</td>
<td>201</td>
</tr>
<tr>
<td>AUTO_LOAD: Load Imported Modules (Freescale, M2)</td>
<td>201</td>
</tr>
<tr>
<td>CHECKSUM: Checksum Computation (ELF)</td>
<td>202</td>
</tr>
<tr>
<td>CHECKKEYS: Check Module Keys (Freescale, M2)</td>
<td>205</td>
</tr>
<tr>
<td>DATA: Specify the RAM Start (Freescale)</td>
<td>205</td>
</tr>
<tr>
<td>DEPENDENCY: Dependency Control</td>
<td>206</td>
</tr>
<tr>
<td>ROOT Keyword</td>
<td>206</td>
</tr>
<tr>
<td>USES Keyword</td>
<td>207</td>
</tr>
<tr>
<td>ADDUSE Keyword</td>
<td>208</td>
</tr>
<tr>
<td>DELUSE Keyword</td>
<td>209</td>
</tr>
<tr>
<td>Overlapping of Local Variables and Parameters</td>
<td>209</td>
</tr>
<tr>
<td>ENTRIES: List of Objects to Link with Application</td>
<td>210</td>
</tr>
<tr>
<td>ELF Specific Issues</td>
<td>211</td>
</tr>
<tr>
<td>HAS_BANKED_DATA: Application Has Banked Data (Freescale)</td>
<td>211</td>
</tr>
<tr>
<td>HEXFILE: Link Hex File with Application</td>
<td>212</td>
</tr>
<tr>
<td>INIT: Specify Application Init Point</td>
<td>213</td>
</tr>
<tr>
<td>LINK: Specify Name of Output File</td>
<td>213</td>
</tr>
<tr>
<td>MAIN: Name of Application Root Function</td>
<td>215</td>
</tr>
<tr>
<td>MAPFILE: Configure Map File Content</td>
<td>215</td>
</tr>
<tr>
<td>NAMES: List Files Building the Application</td>
<td>218</td>
</tr>
<tr>
<td>OVERLAP_GROUP: Application Uses Overlapping (ELF)</td>
<td>219</td>
</tr>
<tr>
<td>PLACEMENT: Place Sections into Segments</td>
<td>221</td>
</tr>
</tbody>
</table>
Table of Contents

PRESTART: Application Prestart Code (Freescale) .......................... 223
SECTIONS: Define Memory Map (Freescale) ................................. 223
SEGMENTS: Define Memory Map (ELF) ................................. 227
STACKSIZE: Define Stack Size ........................................... 234
STACKTOP: Define Stack Pointer Initial Value .......................... 236
START: Specify the ROM Start (Freescale) ............................ 237
VECTOR: Initialize Vector Table ........................................... 237

8 ELF Sections ........................................................................ 241
Segments and Sections ......................................................... 241
Section ............................................................................. 241
Predefined Sections ......................................................... 242

9 Segments ............................................................................ 245
Segments and Sections ......................................................... 245
Segment ............................................................................. 245
Predefined Segments ......................................................... 246

10 Examples of Using Sections ................................................ 249
Example 1 ........................................................................... 249
Example 2 ........................................................................... 249

11 Program Startup ................................................................. 251
Startup Descriptor (ELF) ...................................................... 251
User-Defined Startup Structure: (ELF) .................................. 255
User-Defined Startup Routines (ELF) .................................... 256
Startup Descriptor (Freescale) ................................................ 256
User-Defined Startup Routines (Freescale) .............................. 258
Example of Startup Code in ANSI-C ...................................... 258

12 The Map File ........................................................................ 265
Map File Contents ............................................................... 265

13 ROM Libraries ................................................................. 267
Creating a ROM Library ....................................................... 267
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM Libraries and Overlapping Locals</td>
<td>268</td>
</tr>
<tr>
<td>Using ROM Libraries</td>
<td>268</td>
</tr>
<tr>
<td>Suppressing Initialization</td>
<td>268</td>
</tr>
<tr>
<td><strong>14 How To...</strong></td>
<td></td>
</tr>
<tr>
<td>How To Initialize the Vector Table</td>
<td>275</td>
</tr>
<tr>
<td>Initializing the Vector Table in the SmartLinker prm File</td>
<td>275</td>
</tr>
<tr>
<td>Initializing the Vector Table in the Assembly Source File Using a Relocatable Section</td>
<td>277</td>
</tr>
<tr>
<td>Initializing the Vector Table in the Assembly Source File Using an Absolute Section</td>
<td>280</td>
</tr>
<tr>
<td><strong>II Burner Utility</strong></td>
<td></td>
</tr>
<tr>
<td>Introduction</td>
<td>283</td>
</tr>
<tr>
<td>Product Highlights</td>
<td>284</td>
</tr>
<tr>
<td>Starting the Burner Utility</td>
<td>284</td>
</tr>
<tr>
<td><strong>15 Interactive Burner (GUI)</strong></td>
<td></td>
</tr>
<tr>
<td>Burner Default Configuration Window</td>
<td>287</td>
</tr>
<tr>
<td>Burner Dialog Box</td>
<td>288</td>
</tr>
<tr>
<td>Input/Output Tab</td>
<td>288</td>
</tr>
<tr>
<td>Content Tab</td>
<td>292</td>
</tr>
<tr>
<td>Command File Tab</td>
<td>294</td>
</tr>
<tr>
<td><strong>16 Batch Burner</strong></td>
<td></td>
</tr>
<tr>
<td>Batch Burner User Interface</td>
<td>297</td>
</tr>
<tr>
<td>Syntax of Burner Command Files</td>
<td>298</td>
</tr>
<tr>
<td>Command File Comments</td>
<td>299</td>
</tr>
<tr>
<td>Batch Burner with Makefile</td>
<td>300</td>
</tr>
<tr>
<td>Command File Examples</td>
<td>301</td>
</tr>
<tr>
<td>Parameters of the Command File</td>
<td>303</td>
</tr>
<tr>
<td>baudRate: Baudrate for Serial Communication</td>
<td>304</td>
</tr>
</tbody>
</table>
Table of Contents

busWidth: Data Bus Width ............................................. 304
CLOSE: Close Open File or Communication Port .................. 305
dataBit: Number of Data Bits ....................................... 306
destination: Destination Offset .................................... 306
DO: For Loop Statement List ....................................... 307
ECHO: Echo String onto Output Window .......................... 308
ELSE: Else Part of If Condition .................................. 308
END: For Loop End or If End .................................... 309
FOR: For Loop ...................................................... 310
format: Output Format ............................................. 311
header: Header File for PROM Burner ............................ 312
IF: If Condition ............................................... 312
len: Length to be Copied ......................................... 313
OPENCOM: Open Output Communication Port .................... 314
OPENFILE: Open Output File ..................................... 315
origin: EEPROM Start Address .................................. 316
parity: Set Communication Parity ................................. 316
SENDBYTE: Transfer Bytes ...................................... 317
SENDWORD: Transfer Words .................................... 318
SLINELEN: SRecord Line Length ................................. 319
SRECORD: S-Record Type ....................................... 320
swapByte: Swap Bytes ........................................... 321
THEN: Statementlist for If Condition ............................ 322
TO: For Loop End Condition .................................... 323
undefByte: Fill Byte for Binary Files ......................... 324
PAUSE: Wait until Key Pressed .................................. 324

17 Burner Options .................................................. 327
Burner Option Details ............................................ 328
-D: Display Dialog Box ......................................... 329
-Env: Set Environment Variable ................................ 330
-F: Execute Command File ...................................... 330
-H: Short Help ................................................... 331
-Lic: License Information ...................................... 332
-LicA: License Information about Every Feature in Directory .... 333
Table of Contents

- LicBorrow: Borrow License Feature .............................................. 333
- LicWait: Wait for Floating License from Floating License Server ........ 335
- N: Display Notify Box ................................................................. 336
- NoBeep: No Beep in Case of an Error ........................................... 337
- NoEnv: Do Not Use Environment .................................................. 337
- Ns: Configure S-Records ............................................................ 338
- Prod: Specify Project File at Startup .......................................... 339
- V: Prints Version Information ..................................................... 340
- View: Application Standard Occurrence ..................................... 340
- W: Display Window ................................................................. 341
- Wmsg8x3: Cut File Names in Microsoft Format to 8.3 ..................... 342
- WErrFile: Create “err.log” Error File .......................................... 343
- WmsgCE: RGB Color for Error Messages ...................................... 344
- WmsgCF: RGB Color for Fatal Messages ...................................... 345
- WmsgCl: RGB Color for Information Messages ............................... 345
- WmsgCU: RGB Color for User Messages ....................................... 346
- WmsgCW: RGB Color for Warning Messages .................................. 347
- WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode ................................................................. 348
- WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode ................................................................. 349
- WmsgFob: Message Format for Batch Mode ..................................... 351
- WmsgFoi: Message Format for Interactive Mode .............................. 352
- WmsgFonf: Message Format for No File Information ...................... 353
- WmsgFonp: Message Format for No Position Information ............... 355
- WmsgNe: Number of Error Messages .......................................... 356
- WmsgNi: Number of Information Messages ................................... 357
- WmsgNu: Disable User Messages ................................................ 357
- WmsgNw: Number of Warning Messages ...................................... 358
- WmsgSd: Setting a Message to Disable ........................................ 359
- WmsgSe: Setting a Message to Error ........................................... 360
- WmsgSi: Setting a Message to Information ................................... 360
- WmsgSw: Setting a Message to Warning ....................................... 361
- WOutFile: Create Error Listing File ........................................... 362
- WStdout: Write to Standard Output ............................................ 363
Table of Contents

-W1: No Information Messages ........................................... 364
-W2: No Information and Warning Messages ......................... 364

18 Environment Variables .............................................. 367
The Current Directory .................................................. 368
Global Initialization File (MCUTOOLS.INI) (PC only) ........... 369
Path ............................................................................ 369
Group ........................................................................... 370
DefaultDir ................................................................. 370
SaveOnExit ................................................................. 371
SaveAppearance ......................................................... 371
SaveEditor ............................................................... 371
SaveOptions ............................................................. 372
RecentProject0, RecentProject1 ....................................... 372
Editor_Name .............................................................. 373
Editor_Exe .............................................................. 373
Editor_Opts ............................................................ 373
Local Configuration File (Usually project.ini) .................... 374
Editor_Name .............................................................. 375
Editor_Exe .............................................................. 376
Editor_Opts ............................................................ 376
RecentCommandLineX, X= integer .................................... 377
CurrentCommandLine .................................................. 377
StatusBarEnabled ...................................................... 377
ToolbarEnabled .......................................................... 378
WindowPos .............................................................. 378
WindowFont ............................................................. 379
TipFilePos .............................................................. 379
ShowTipOfDay ........................................................... 380
Options ................................................................. 380
EditorType .............................................................. 380
EditorCommandLine ..................................................... 381
EditorDDEClientName ................................................. 381
EditorDDETtopicName ................................................ 382
EditorDDEServiceName .............................................. 382
### Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BurnerUndefByte</td>
<td>383</td>
</tr>
<tr>
<td>BurnerSwapByte</td>
<td>383</td>
</tr>
<tr>
<td>BurnerOrigin</td>
<td>384</td>
</tr>
<tr>
<td>BurnerDestination</td>
<td>384</td>
</tr>
<tr>
<td>BurnerLength</td>
<td>384</td>
</tr>
<tr>
<td>BurnerFormat</td>
<td>385</td>
</tr>
<tr>
<td>BurnerDataBus</td>
<td>385</td>
</tr>
<tr>
<td>BurnerOutputType</td>
<td>386</td>
</tr>
<tr>
<td>BurnerDataBits</td>
<td>386</td>
</tr>
<tr>
<td>BurnerParity</td>
<td>387</td>
</tr>
<tr>
<td>BurnerByteCommands</td>
<td>387</td>
</tr>
<tr>
<td>BurnerBaudRate</td>
<td>388</td>
</tr>
<tr>
<td>BurnerOutputFile</td>
<td>388</td>
</tr>
<tr>
<td>BurnerHeaderFile</td>
<td>388</td>
</tr>
<tr>
<td>BurnerInputFile</td>
<td>389</td>
</tr>
<tr>
<td>Paths</td>
<td>390</td>
</tr>
<tr>
<td>Line Continuation</td>
<td>391</td>
</tr>
<tr>
<td>Environment Variable Details</td>
<td>392</td>
</tr>
<tr>
<td>DEFAULTDIR: Default Current Directory</td>
<td>393</td>
</tr>
<tr>
<td>ENVIRONMENT: Environment File Specification</td>
<td>394</td>
</tr>
<tr>
<td>ERRORFILE: Error File Name Specification</td>
<td>395</td>
</tr>
<tr>
<td>GENPATH: #include “File” Path</td>
<td>397</td>
</tr>
<tr>
<td>TMP: Temporary Directory</td>
<td>398</td>
</tr>
<tr>
<td><strong>19 Burner Messages</strong></td>
<td>399</td>
</tr>
<tr>
<td>Message Kinds</td>
<td>399</td>
</tr>
<tr>
<td>Information</td>
<td>399</td>
</tr>
<tr>
<td>WARNING</td>
<td>399</td>
</tr>
<tr>
<td>ERROR</td>
<td>399</td>
</tr>
<tr>
<td>FATAL</td>
<td>399</td>
</tr>
<tr>
<td>DISABLE</td>
<td>399</td>
</tr>
<tr>
<td>Message Details</td>
<td>400</td>
</tr>
<tr>
<td>Message List</td>
<td>401</td>
</tr>
<tr>
<td>B1: Unknown Message Occurred</td>
<td>401</td>
</tr>
<tr>
<td>B2: Message Overflow, Skipping &lt;kind&gt; Messages</td>
<td>401</td>
</tr>
</tbody>
</table>
Table of Contents

B50: Input file ‘<file>’ not found . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .401
B51: Cannot Open Statistic Log File <file> . . . . . . . . . . . . . . . . . . . . . . . .402
B52: Error in Command Line '<cmd>' . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .402
B64: Line Continuation Occurred in <FileName> . . . . . . . . . . . . . . . . . . .402
B65: Environment Macro Expansion Error '<description>' for
    <variablename> . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .403
B66: Search Path <Name> Does Not Exist . . . . . . . . . . . . . . . . . . . . . . . . .404
B1000: Could Not Open '<FileType>' '<File>. . . . . . . . . . . . . . . . . . . . . .404
B1001: Error in Input File Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .405
B1002: Selected Communication Port is Busy . . . . . . . . . . . . . . . . . . . . . .405
B1003: Timeout or Failure for the Selected Communication . . . . . . . . . . .405
B1004: Error in Macro '<macro>' at Position <pos>: '<msg>' . . . . . . . . . .406
B1005: Error in Command Line at Position <pos>: '<msg>' . . . . . . . . . . .406
B1006: '<msg>' . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .406

III Libmaker

Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .407
Product Highlights. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .407
Starting the Libmaker Utility . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .408
User Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .408

20 Libmaker Interface 411

Startup Command Line Options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .411
Command Line Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .411
Libmaker Commands. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .411
Managing Libraries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .412
Libmaker Graphic User Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .415
Libmaker Default Configuration Window . . . . . . . . . . . . . . . . . . . . . . . . . . . .416
Default Configuration Window Status Bar . . . . . . . . . . . . . . . . . . . . . . . . . .418
Configuration Window . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .423
Libmaker Option Settings Window . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .434
Libmaker Message Settings Window . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .436
Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>About Libmaker Dialog Box</td>
<td>439</td>
</tr>
<tr>
<td>Libmaker Environment Variables</td>
<td>440</td>
</tr>
<tr>
<td>Local Configuration File (Usually project.ini)</td>
<td>441</td>
</tr>
<tr>
<td>The Current Directory</td>
<td>441</td>
</tr>
<tr>
<td>Paths</td>
<td>442</td>
</tr>
<tr>
<td>Line Continuation</td>
<td>443</td>
</tr>
<tr>
<td>Environment Variable Details</td>
<td>444</td>
</tr>
<tr>
<td>DEFAULTDIR: Current Directory</td>
<td>444</td>
</tr>
<tr>
<td>ENVIRONMENT: Environment File Specification</td>
<td>445</td>
</tr>
<tr>
<td>ERRORFILE: Error File Name Specification</td>
<td>446</td>
</tr>
<tr>
<td>GENPATH: Defines Paths to search for input Files</td>
<td>448</td>
</tr>
<tr>
<td>TEXTPATH: Text Path</td>
<td>449</td>
</tr>
<tr>
<td>TMP: Temporary Directory</td>
<td>449</td>
</tr>
<tr>
<td>21 Libmaker Options</td>
<td>451</td>
</tr>
<tr>
<td>Options</td>
<td>451</td>
</tr>
<tr>
<td>Option Details</td>
<td>451</td>
</tr>
<tr>
<td>-Cmd: Libmaker Commands</td>
<td>453</td>
</tr>
<tr>
<td>-Env: Set Environment Variable</td>
<td>454</td>
</tr>
<tr>
<td>-H: Short Help</td>
<td>455</td>
</tr>
<tr>
<td>-Lic: License Information</td>
<td>456</td>
</tr>
<tr>
<td>-LicA: License Information About Every Feature in Directory</td>
<td>457</td>
</tr>
<tr>
<td>-LicBorrow: Borrow License Feature</td>
<td>458</td>
</tr>
<tr>
<td>-LicWait: Wait Until Floating License Available from Floating License Server</td>
<td>459</td>
</tr>
<tr>
<td>-Mar: Freescale Archive Commands</td>
<td>460</td>
</tr>
<tr>
<td>-N: Display Notify Box (PC Only)</td>
<td>460</td>
</tr>
<tr>
<td>-NoBeep: No Beep in Case of an Error</td>
<td>461</td>
</tr>
<tr>
<td>-NoPath: Strip Path Info</td>
<td>462</td>
</tr>
<tr>
<td>-Prod: Specify Project File at Startup</td>
<td>463</td>
</tr>
<tr>
<td>-V: Prints the Libmaker Version</td>
<td>463</td>
</tr>
<tr>
<td>-View Application Standard Occurrence (PC Only)</td>
<td>464</td>
</tr>
<tr>
<td>-W1: No Information Messages</td>
<td>465</td>
</tr>
<tr>
<td>-W2: No Information and Warning Messages</td>
<td>466</td>
</tr>
<tr>
<td>-Wmsg8x3: Cut File Names in Microsoft Format to 8.3 (PC Only)</td>
<td>467</td>
</tr>
</tbody>
</table>
Table of Contents

-WErrFile: Create “err.log” Error File. .........................468
-WmsgCE: RGB Color for Error Messages ......................469
-WmsgCF: RGB Color for Fatal Messages ......................469
-WmsgCI: RGB Color for Information Messages ...............470
-WmsgCU: RGB Color for User Messages .......................471
-WmsgCW: RGB Color for Warning Messages ...................471
-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode ........................................472
-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode 474
-WmsgFob: Message Format for Batch Mode ....................475
-WmsgFoi: Message Format for Interactive Mode ..............477
-WmsgFonf: Message Format for No File Information ..........478
-WmsgFonp: Message Format for No Position Information ..........479
-WmsgNe: Number of Error Messages ..............................480
-WmsgNi: Number of Information Messages ....................481
-WmsgNu: Disable User Messages .................................482
-WmsgNw: Number of Warning Messages ..........................483
-WmsgSd: Disabling a Message ..................................484
-WmsgSe: Setting Message Type to Error ..................484
-WmsgSi: Set Message Type to Information ..................485
-WmsgSw: Setting Message Type to Warning ..................486
-WOutFile: Create Error List File ..............................487
-WStdout: Create Error List File ..............................488

22 Libmaker Messages 489

Message Types .........................................................489
INFORMATION .........................................................489
WARNING ............................................................489
ERROR ...............................................................489
FATAL ...............................................................489
DISABLE ............................................................489
Message Details .......................................................490
Message List ........................................................490
LM1: Unknown Message Occurred .................................490
<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM2: ................. Message Overflow, Skipping &lt;kind&gt; Messages 491</td>
</tr>
<tr>
<td>LM50: Input File ‘&lt;file&gt;’ Not Found .......................... 491</td>
</tr>
<tr>
<td>LM51: Cannot Open Statistic Log File &lt;file&gt; ......................... 491</td>
</tr>
<tr>
<td>LM52: Error in Command Line &lt;cmd&gt; ................................... 492</td>
</tr>
<tr>
<td>LM64: Line Continuation Occurred in &lt;FileName&gt; ...................... 492</td>
</tr>
<tr>
<td>LM65: Environment Macro Expansion Message '&lt;description&gt;' for &lt;variablename&gt; ................................................. 493</td>
</tr>
<tr>
<td>LM66: Search Path &lt;Name&gt; Does Not Exist .............................. 494</td>
</tr>
<tr>
<td>23 Environment Variables ................................................. 495</td>
</tr>
<tr>
<td>Directories .................................................................... 495</td>
</tr>
<tr>
<td>Other Environment Variables ............................................. 496</td>
</tr>
<tr>
<td>ERRORFILE: Error File Name Specification ............................ 496</td>
</tr>
<tr>
<td>24 EBNF Notation ............................................................ 499</td>
</tr>
<tr>
<td>Introduction to EBNF ...................................................... 499</td>
</tr>
<tr>
<td>IV Decoder ..................................................................... 503</td>
</tr>
<tr>
<td>Introduction .................................................................... 503</td>
</tr>
<tr>
<td>Product Highlights .......................................................... 503</td>
</tr>
<tr>
<td>User Interface .................................................................. 504</td>
</tr>
<tr>
<td>25 Decoder Environment .................................................... 505</td>
</tr>
<tr>
<td>Settings ........................................................................ 505</td>
</tr>
<tr>
<td>Paths ............................................................................ 506</td>
</tr>
<tr>
<td>Line Continuation ............................................................ 507</td>
</tr>
<tr>
<td>Environment Variables ...................................................... 508</td>
</tr>
<tr>
<td>DEFAULTDIR: Current Directory ........................................... 508</td>
</tr>
<tr>
<td>ENVIRONMENT: Environment File Specification ...................... 509</td>
</tr>
<tr>
<td>GENPATH: Defines Paths to Search for Input Files .................... 510</td>
</tr>
<tr>
<td>TEXTPATH: Text Path .......................................................... 511</td>
</tr>
</tbody>
</table>
# Table of Contents

## 26 Input and Output Files 513

<table>
<thead>
<tr>
<th>Input Files</th>
<th>.513</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute Files</td>
<td>.513</td>
</tr>
<tr>
<td>Object File</td>
<td>.513</td>
</tr>
<tr>
<td>S-Record Files</td>
<td>.514</td>
</tr>
<tr>
<td>Intel Hex Files</td>
<td>.514</td>
</tr>
<tr>
<td>Output Files</td>
<td>.514</td>
</tr>
</tbody>
</table>

## 27 Decoder Options 517

<table>
<thead>
<tr>
<th>Using Decoder Options</th>
<th>.517</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option Topics</td>
<td>.517</td>
</tr>
<tr>
<td>Special Modifiers</td>
<td>.518</td>
</tr>
<tr>
<td>-A: Print Full Listing</td>
<td>.519</td>
</tr>
<tr>
<td>-C: Write Disassembly Listing With Source Code</td>
<td>.520</td>
</tr>
<tr>
<td>-D: Decode DWARF Sections</td>
<td>.521</td>
</tr>
<tr>
<td>-E: Decode ELF sections</td>
<td>.524</td>
</tr>
<tr>
<td>-Ed: Dump ELF Sections in LST File</td>
<td>.526</td>
</tr>
<tr>
<td>-Env: Set Environment Variable</td>
<td>.527</td>
</tr>
<tr>
<td>-F: Object File Format</td>
<td>.527</td>
</tr>
<tr>
<td>-H: Prints the List of All Available Options</td>
<td>.528</td>
</tr>
<tr>
<td>-L: Produce Inline Assembly File</td>
<td>.529</td>
</tr>
<tr>
<td>-Lic: Print License Information</td>
<td>.530</td>
</tr>
<tr>
<td>-LicA: License Information About Every Feature in Directory</td>
<td>.530</td>
</tr>
<tr>
<td>-LicBorrow: Borrow License Feature</td>
<td>.531</td>
</tr>
<tr>
<td>-LicWait: Wait for Floating License from Floating License Server</td>
<td>.532</td>
</tr>
<tr>
<td>-N: Display Notify Box</td>
<td>.532</td>
</tr>
<tr>
<td>-NoBeep: No Beep in Case of an Error</td>
<td>.533</td>
</tr>
<tr>
<td>-NoEnv: Do not use Environment</td>
<td>.534</td>
</tr>
<tr>
<td>-NoSym: No Symbols in Disassembled Listing</td>
<td>.534</td>
</tr>
<tr>
<td>-O: Defines Listing File Name</td>
<td>.535</td>
</tr>
<tr>
<td>-Proc: Set Processor</td>
<td>.536</td>
</tr>
<tr>
<td>-T: Show Cycle Count for Each Instruction</td>
<td>.537</td>
</tr>
<tr>
<td>-V: Print Decoder Version</td>
<td>.538</td>
</tr>
<tr>
<td>-View: Application Standard Occurrence (PC)</td>
<td>.538</td>
</tr>
</tbody>
</table>
28 Decoder Messages

Types of Generated Messages ........................................... 561
Message Details .......................................................... 562
List of Messages .......................................................... 563
D1: .......................................................... Unknown Message Occurred563
D2: .......................................................... Message Overflow, Skipping <kind> Messages563
D50: .......................................................... Input File ‘<file>’ Not Found563
Table of Contents

D51: ........................................ Cannot Open Statistic Log File <file>564
D52: ........................................ Error in Command Line <cmd>564
D64: ........................................ Line Continuation Occurred in <FileName>564
D65: ................ Environment Macro Expansion Message '<description>' for <variablename> ...............................565
D66: ........................................ Search Path <Name> Does Not Exist566
D1000: .............................. Bad Hex Input File <Description>566
D1001: .............. Because Current Processor is Unknown, No Disassembly is Generated. Use -proc................566

29 Decoder Controls 567
Pull-Down Menus ..................................................567
File Menu .............................................................568
Decoder Menu ......................................................570
View Menu ..........................................................571
Help Menu ..........................................................572
Graphical User Interface .........................................572
Decoder Main Window ............................................573
Decoder Configuration Window .................................575
Decoder Option Settings .........................................582
About Decoder Dialog Box .......................................585
Specifying the Input File .........................................586
Use the Command Line in the Tool Bar to Decode ..........586
Processing a File Already Run ...................................586
Message and Error Feedback ....................................587
Using Information from the Main Window .....................587
Using a User-defined Editor .....................................587

V Maker: The Make Tool

30 Maker Controls 591
Graphical User Interface ........................................591
Maker Main Window ...............................................591
## 31 Using Maker

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Making Modula–2 Applications</td>
<td>615</td>
</tr>
<tr>
<td>Making C Applications</td>
<td>615</td>
</tr>
<tr>
<td>Using Makefiles</td>
<td>616</td>
</tr>
<tr>
<td>User-defined Macros (Static Macros)</td>
<td>618</td>
</tr>
<tr>
<td>Definition</td>
<td>618</td>
</tr>
<tr>
<td>Reference</td>
<td>618</td>
</tr>
<tr>
<td>Redefinition</td>
<td>618</td>
</tr>
<tr>
<td>Macro Substitution</td>
<td>618</td>
</tr>
<tr>
<td>Macros &amp; Comments</td>
<td>619</td>
</tr>
<tr>
<td>Concatenation</td>
<td>620</td>
</tr>
<tr>
<td>Command-Line Macros</td>
<td>620</td>
</tr>
<tr>
<td>Dynamic Macros</td>
<td>621</td>
</tr>
<tr>
<td>Inference Rules</td>
<td>622</td>
</tr>
<tr>
<td>Multiple Inference Rules</td>
<td>623</td>
</tr>
<tr>
<td>Directives and Special Targets</td>
<td>624</td>
</tr>
<tr>
<td>Built-In Commands</td>
<td>625</td>
</tr>
<tr>
<td>Command Line</td>
<td>627</td>
</tr>
<tr>
<td>Implementation Restrictions</td>
<td>627</td>
</tr>
</tbody>
</table>
# Table of Contents

## 32 Maker Environment Variables

- Setting Parameters ........................................ 629
- Current Directory ......................................... 630
- Global Initialization File (MCUTOOLS.INI) .............. 631
- [Installation] Section .................................. 631
- [Options] Section ........................................ 632
- [Editor] Section .......................................... 632
- Local Configuration File (Usually project.ini) ........ 634
- [Editor] Section ........................................ 635
- Line Continuation ........................................ 637
- Input and Output Files .................................... 638
- Error Listing ............................................. 638
- List of Environment Variables ......................... 639
  - COMP: Modula-2 Compiler .............................. 639
  - DEFAULTDIR: Default Current Directory ............ 640
  - ENVIRONMENT: Environment File Specification .... 641
  - ERRORFILE: Error File Name Specification ......... 642
  - FLAGS: Options for Modula-2 Compiler ............... 643
  - GENPATH: #include “File” Path ....................... 644
  - LINK: Linker for Modula-2 ............................ 645
  - TEXTFAMILY: Text Font Family ....................... 645
  - TEXTKIND: Text Font Character Set .................. 646
  - TEXTSIZE: Text Font Size ............................. 647
  - TEXTSTYLE: Text Font Style ......................... 648

## 33 Building Libraries

- Maker Directory Structure ............................... 649
- Configuring WinEdit for the Maker .................... 650
- Configuring default.env for the Maker ................. 651
- Building Libraries with Defined Memory Model Options 651
- Building Libraries With Objects Added ................. 652
- Structured Makefiles for Libraries .................... 654
Table of Contents

34 Maker Options 657

Option Groups ................................................................. 657
Option Details ................................................................. 658

-A: Warning for Missing .DEF File ................................. 658
-C: Ignore Case ................................................................. 659
-D: Define a Macro .............................................................. 659
-Disp: Display Mode ............................................................. 660
-E: Unknown Macros as Empty Strings ......................... 660
-Env: Set Environment Variable ........................................... 661
-H: Short Help ................................................................. 662
-I: Ignore Exit Codes ......................................................... 662
-L: List Modules ................................................................. 663
-Lic: License Information ................................................... 663
-LicA: License Information About Every Feature in Directory 664
-LicBorrow: Borrow License Feature ................................. 664
-LicWait: Wait Until Floating License Is Available from Floating License Server ............................................. 665
-M: Produce Make File ......................................................... 666
-MkAll: Make Always ......................................................... 666
-N: Display Notify Box .......................................................... 667
-NoBeep: No Beep in Case of an Error............................... 667
-NoCapture: Do Not Redirect stdout of Called Processes ...... 668
-NoEnv: Do Not Use Environment .......................................... 668
-O: Compile Only ................................................................. 669
-S: Silent Mode ................................................................. 670
-V: Prints the Version .......................................................... 670
-View: Application Standard Occurrence (PC) ................. 671
-WErrFile: Create “err.log” Error File ............................... 672
-Wmsg8x3: Cut File Names in Microsoft Format to 8.3 ........ 672
-WmsgCE: RGB Color for Error Messages ......................... 673
-WmsgCF: RGB Color for Fatal Messages ......................... 674
-WmsgCI: RGB Color for Information Messages ................. 674
-WmsgCU: RGB Color for User Messages ......................... 675
-WmsgCW: RGB Color for Warning Messages ..................... 676
### Table of Contents

- **WmsgFb**: Set Message File Format for Batch Mode .................. 676
- **WmsgFi**: Set Message Format for Interactive Mode ............... 677
- **WmsgFob**: Message Format for Batch Mode .................. 678
- **WmsgFoi**: Message Format for Interactive Mode ............... 679
- **WmsgFonf**: Message Format for No File Information ............. 681
- **WmsgFonp**: Message Format for No Position Information ........ 682
- **WmsgNe**: Number of Error Messages ................................ 683
- **WmsgNi**: Number of Information Messages ....................... 683
- **WmsgNu**: Disable User Messages .................................. 684
- **WmsgNw**: Number of Warning Messages ............................ 685
- **WmsgSd**: Setting a Message to Disable ............................. 686
- **WmsgSe**: Setting a Message to Error .............................. 686
- **WmsgSi**: Setting a Message to Information ....................... 687
- **WmsgSw**: Setting a Message to Warning ............................ 687
- **WmsgVrb**: Verbose Mode ........................................... 688
- **WOutFile**: Create Error Listing File .............................. 689
- **WStdout**: Write to Standard Output ............................... 689
- **W1**: No Information Messages ..................................... 690
- **W2**: No Information and Warning Messages ....................... 691

### 35 Maker Messages

Kinds of Maker Messages ............................................. 693

Makefile Messages ....................................................... 694

- **M1**: Unknown Message Occurred .................................. 694
- **M2**: Message Overflow, Skipping <kind> Messages ............. 694
- **M50**: Input File ‘<file>’ Not Found ............................... 695
- **M51**: Cannot Open Statistic Log File <file> .................... 695
- **M64**: Line Continuation Occurred in <FileName> ............... 696
- **M65**: Environment Macro Expansion Error ’<description>’ for
  <variablename> ....................................................... 697
- **M66**: Search Path <Name> Does Not Exist ....................... 697
- **M5000**: User Requested Stop ...................................... 698
- **M5001**: Error in Command Line ................................. 698
- **M5002**: Can’t Return to <makefile> at End of Include File .... 699
- **M5003**: Illegal Dependency ........................................ 699
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>M5004</td>
<td>Illegal Macro Reference</td>
<td>700</td>
</tr>
<tr>
<td>M5005</td>
<td>Macro Substitution Too Complex</td>
<td>700</td>
</tr>
<tr>
<td>M5006</td>
<td>Macro Reference Not Closed</td>
<td>701</td>
</tr>
<tr>
<td>M5007</td>
<td>Unknown Macro: <code>&lt;macroname&gt;</code></td>
<td>701</td>
</tr>
<tr>
<td>M5008</td>
<td>Macro Definition or Command Line Too Long</td>
<td>701</td>
</tr>
<tr>
<td>M5009</td>
<td>Illegal Include Directive</td>
<td>702</td>
</tr>
<tr>
<td>M5010</td>
<td>Illegal Line</td>
<td>702</td>
</tr>
<tr>
<td>M5011</td>
<td>Illegal Suffix for Inference Rule</td>
<td>703</td>
</tr>
<tr>
<td>M5012</td>
<td>Include File Not Found: <code>&lt;includefile&gt;</code></td>
<td>703</td>
</tr>
<tr>
<td>M5013</td>
<td>Include File Too Long: <code>&lt;includefile&gt;</code></td>
<td>704</td>
</tr>
<tr>
<td>M5014</td>
<td>Circular Macro Substitution in <code>&lt;macroname&gt;</code></td>
<td>704</td>
</tr>
<tr>
<td>M5015</td>
<td>Colon (:) Expected</td>
<td>704</td>
</tr>
<tr>
<td>M5016</td>
<td>Filename After INCLUDE Expected</td>
<td>705</td>
</tr>
<tr>
<td>M5017</td>
<td>Circular Include, File <code>&lt;includefile&gt;</code></td>
<td>705</td>
</tr>
<tr>
<td>M5018</td>
<td>Entry Doesn't Start at Column 0</td>
<td>705</td>
</tr>
<tr>
<td>M5019</td>
<td>No Makefile Found</td>
<td>706</td>
</tr>
<tr>
<td>M5020</td>
<td>Fatal Error During Initialization</td>
<td>706</td>
</tr>
<tr>
<td>M5021</td>
<td>Nothing to Make: No Target Found</td>
<td>706</td>
</tr>
<tr>
<td>M5022</td>
<td>Don't Know How to Make <code>&lt;target&gt;</code></td>
<td>707</td>
</tr>
<tr>
<td>M5023</td>
<td>Circular Dependencies Between <code>&lt;target1&gt;</code> and <code>&lt;target2&gt;</code></td>
<td>707</td>
</tr>
<tr>
<td>M5024</td>
<td>Illegal Option</td>
<td>708</td>
</tr>
<tr>
<td>M5027</td>
<td>Making Target <code>&lt;target&gt;</code></td>
<td>708</td>
</tr>
<tr>
<td>M5028</td>
<td>Command Line Too Long: <code>&lt;commandline&gt;</code></td>
<td>709</td>
</tr>
<tr>
<td>M5029</td>
<td>Illegal Target Name: <code>&lt;targetname&gt;</code></td>
<td>709</td>
</tr>
<tr>
<td></td>
<td>Exec Process Messages</td>
<td>709</td>
</tr>
<tr>
<td>M5100</td>
<td>Command Line Too Long for Exec</td>
<td>709</td>
</tr>
<tr>
<td>M5101</td>
<td>Two File Names Expected</td>
<td>710</td>
</tr>
<tr>
<td>M5102</td>
<td>Input File Not Found</td>
<td>710</td>
</tr>
<tr>
<td>M5103</td>
<td>Output File Not Opened</td>
<td>710</td>
</tr>
<tr>
<td>M5104</td>
<td>Error While Copying</td>
<td>711</td>
</tr>
<tr>
<td>M5105</td>
<td>Renaming Failed</td>
<td>711</td>
</tr>
<tr>
<td>M5106</td>
<td>File Name Expected</td>
<td>712</td>
</tr>
<tr>
<td>M5107</td>
<td>File Does Not Exist</td>
<td>712</td>
</tr>
<tr>
<td>M5108</td>
<td>Called Application Detected an Error</td>
<td>713</td>
</tr>
<tr>
<td>M5109</td>
<td>Echo <code>&lt;commandline&gt;</code></td>
<td>713</td>
</tr>
</tbody>
</table>
Table of Contents

M5110: Called Application Caused a System Error ......................... 713
M5111: Change Directory (cd) Failed. ..................................... 714
M5112: Called Application: <error> ...................................... 714
M5113: Called Application: <warning> ................................... 715
M5114: Called Application: <information> ................................. 715
M5115: Called Application: <fatal> ...................................... 716
M5116: Could Not Delete File ............................................. 717
M5117: Path Was Not Found. ............................................... 717
M5118: Could Not Create Process: <diagnostic> .......................... 717
M5119: Exec <commandline> .............................................. 718
M5120: Running Version with Limited Number of Execution Calls. Number of Allowed Execution Calls Exceeded. ......................... 718
M5121: The Files <file1> and <file2> Are Not Identical ................ 718
M5122: The Files <file1> and <file2> Are Identical ...................... 719
M5153: Processing Make Files Under Win32s Is Not Supported by the Maker ........................................... 720

Modula-2 Maker Messages .................................................... 720
M5700: Environment Variable COMP Not Set ............................... 720
M5701: Environment Variable LINK Not Set ............................... 720
M5702: Neither Source Nor Symbol File Found: <source file> ............ 720
M5703: Circular Imports in Definition Modules. .......................... 721
M5704: Can't Recompile <source file> (No Source Found) ............... 721
M5705: No Make File Generated (Top Module Not Found) ................ 722
M5706: Couldn't Open the Listing File <list file> ....................... 722
M5708: Couldn't Open the Makefile ....................................... 723
M5761: Wrote Makefile <makefile> ....................................... 723
M5762: Wrote Listing File <listfile> .................................... 723
M5763: Compilation Sequence ............................................. 724

36 Using the Linux Command Line Programs .............................. 725
Command Line Arguments ..................................................... 725
Command Examples .......................................................... 725
Using a Makefile ............................................................. 727

Index .................................................................................. 729
Introduction

CodeWarrior IDE Utilities

The HC(S)08, RS08 and HC(S)12 Build Tools Utility Manual describes the following five CodeWarrior IDE utilities:

SmartLinker

The CodeWarrior IDE SmartLinker utility merges the various object files of an application into one absolute file (or .ABS file) that can be converted to a S-Record or an Intel Hex file using the Burner program or loaded into the target using the Downloader/Debugger.

This utility is a “smart linker”, linking only those objects that are actually used by your application. This linker is able to generate either Freescale (former Hiware) or ELF absolute files.

Burner Utility

The CodeWarrior IDE burner utility converts an .ABS file into a file that can be handled by an EPROM burner.

Libmaker

The CodeWarrior IDE Libmaker is a utility program for creating and maintaining object file libraries.

Decoder

The CodeWarrior IDE ELF/Freescale (former Hiware) Decoder utility disassembles object files, absolute files and libraries in the Freescale (former Hiware) object file format or ELF/DWARF format, along with S-Record files.

Maker: The Make Tool

The CodeWarrior IDE Maker Utility implements the UNIX make command with a Graphical User Interface (GUI). In addition, you can use Maker to build Modula-2 applications as well as maintain C/C++ projects.
Starting a CodeWarrior Utility

All of the utilities described in this book may be started from executable files located in the Prog folder of your CodeWarrior IDE installation. The executable files are:

- `maker.exe` Maker: The Make Tool
- `burner.exe` The Burner Utility
- `decoder.exe` The Decoder
- `libmaker.exe` Libmaker
- `linker.exe` SmartLinker

With a standard full installation of the HC(S)08/RS08 CodeWarrior IDE, the executable files are located at:

C:\Program Files\Freescale\CW08 v5.x\Prog

To start any CodeWarrior Utility, you can click on the appropriate executable file (.exe).
SmartLinker

This chapter describes the SmartLinker utility. 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 an 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. It will link only those objects that are actually used by your application.

This linker is able to generate either Freescale (former Hiware) or ELF absolute files. For compatibility purpose, the Freescale input syntax is also supported when ELF absolute files are generated.

Manual Notation

Throughout this document, features or syntax which are supported only when ELF/DWARF (DWARF - Debugging With Attribute Record Format) absolute files are generated will be followed by (ELF).

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

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

Product Features

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 the Segments and Sections chapters).
- Linking to objects already allocated in a previous link session (ROM libraries).

**NOTE** The code for application startup is a separate file written in inline assembly and can be easily adapted to your particular needs. In this manual, the startup file is called startup. However, this is a generic file name that has to be replaced by the real target startup file name. Please see also the README.TXT file in the appropriate subdirectory of the installation LIB 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.

Section Contents

This section consists of the following chapters:

- SmartLinker User Interface — Describes the features of the SmartLinker user interface
- Environment Variables — Describes the environment variables used by the SmartLinker
- SmartLinker Files — Describes the input and output files used by the SmartLinker
- SmartLinker Options — Provides detailed descriptions of the full set of SmartLinker options
Starting the SmartLinker Utility

All of the utilities described in this book may be started from executable files located in the Prog folder of your IDE installation. The executable files are:

- maker.exe  Maker: The Make Tool
- burner.exe  The Burner Utility
- decoder.exe  The Decoder
- libmaker.exe  Libmaker
- linker.exe  SmartLinker

With a standard full installation of the HC(S)08/RS08 CodeWarrior IDE, the executable files are located at:

C:\Program Files\Freescale\CW08 v5.x\Prog

- To start the SmartLinker Utility, you can click on linker.exe.
SmartLinker User Interface

The SmartLinker runs under Win32.
Start the linker from the CodeWarrior installation prog folder.

SmartLinker Main Window

The SmartLinker Main window provides a window title, a menu bar, a tool bar, a content area, and a status bar, as shown in Figure 1.1.

Figure 1.1 SmartLinker Main Window
SmartLinker User Interface
SmartLinker Main Window

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 that some values have 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”).

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 available only if a file context is available.

If a file cannot be opened although a context menu entry is present, the editor configuration information is not correct (see the section “Editor Settings Tab”).

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 red, Fatal Errors are dark red, Warnings are blue, and Information Messages are green.

**Main Window Tool Bar**

*Figure 1.2* shows the SmartLinker Main Window Tool Bar buttons.

**Figure 1.2 Tool Bar Icons**

The three icons on the left are linked with the corresponding entries of the File menu. You can use these icons to reset, load, and save configuration files for the linker.

Use the Help icon and the Context Help icon to open the Help file and the Context Help. When pressing the context help button the mouse cursor changes its form and has a question mark beside the arrow. The help is called for the next item that is clicked. Use the Context Help to get specific help on menus, toolbar buttons, or on the window area.

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.

Use the **Stop Linking** button to abort the current link session. If no link session is running, this button is disabled (gray).

Use the **Option Settings** button to open the **Option Settings** dialog.

Use the **Message Settings** button to open the **Message Settings** dialog.

Use the **Clear** button to clear the SmartLinker window content area.

Activate the command line in the toolbar by using the F2 key.

Use the right mouse button to display a context menu.

Messages are colored according to their Message Class.
Main Window Status Bar

shows the SmartLinker Main Window Status bar.

Figure 1.3 Main Window 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.

Main Window Menu Bar

The following menus are available in the menu bar:

Table 1.1 Main Window Menus

<table>
<thead>
<tr>
<th>Menu</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“File Menu”</td>
<td>Contains entries to manage SmartLinker configuration files.</td>
</tr>
<tr>
<td>“SmartLinker Menu”</td>
<td>Contains entries to set SmartLinker options.</td>
</tr>
<tr>
<td>“View Menu”</td>
<td>Contains entries to customize the SmartLinker window output.</td>
</tr>
<tr>
<td>Help</td>
<td>A standard Windows Help menu.</td>
</tr>
</tbody>
</table>

File Menu

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

- SmartLinker option settings specified in the SmartLinker dialog boxes.
- Message settings which specify which messages to display and which to treat as errors.
- List of the last command line executed and the current command line.
- Window position, size and font.
- Tips of the Day settings, including the enable at startup setting and the current entry.

Configuration files are text files, which have the standard extension .ini. You can define as many configuration files as required for your 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 1.2 describes the menu items.
### Table 1.2  File Menu Item Description

<table>
<thead>
<tr>
<th>Menu Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link</td>
<td>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 by clicking OK.</td>
</tr>
<tr>
<td>New/Default Configuration</td>
<td>Resets the SmartLinker option settings to the default values. The SmartLinker options, which are activated per default, are specified in section Command Line Options from this document.</td>
</tr>
<tr>
<td>Load Configuration</td>
<td>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.</td>
</tr>
<tr>
<td>Save Configuration</td>
<td>Saves the current settings in the configuration file specified on the title bar.</td>
</tr>
<tr>
<td>Save Configuration as...</td>
<td>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 by clicking OK.</td>
</tr>
<tr>
<td>Configuration..</td>
<td>Opens the Configuration dialog box to specify the editor used for error feedback, which parts to save with a configuration, and environment variable settings</td>
</tr>
<tr>
<td>1.... project.ini</td>
<td>Recent project list. This list can be accessed to open a recently opened project again.</td>
</tr>
<tr>
<td>2 ....</td>
<td></td>
</tr>
<tr>
<td>Exit</td>
<td>Closes the SmartLinker.</td>
</tr>
</tbody>
</table>
SmartLinker User Interface

SmartLinker Main Window

**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 1.3 describes the SmartLinker menu items.

<table>
<thead>
<tr>
<th>Menu Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Options...</td>
<td>Allows you to define the options which must be activated when linking an input file (see Option Settings Dialog Box below).</td>
</tr>
<tr>
<td>Messages</td>
<td>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).</td>
</tr>
<tr>
<td>Stop Linking</td>
<td>Stops the currently running linking process. This entry is only enabled (black) when a link process currently takes place. Otherwise, it is gray.</td>
</tr>
</tbody>
</table>

**View Menu**

The View menu allows you to customize the linker window. You can specify if the status bar and the tool bar should be displayed or hidden. You can also define the font used in the window or clear the window. Table 1.4 describes the View menu items.

<table>
<thead>
<tr>
<th>Menu Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool Bar</td>
<td>Switches display from the tool bar in the SmartLinker window.</td>
</tr>
<tr>
<td>Status Bar</td>
<td>Switches display from the status bar in the SmartLinker window.</td>
</tr>
<tr>
<td>Log...</td>
<td>Allows you to customize the output in the SmartLinker window content area. The following entries are available when Log... is selected:</td>
</tr>
<tr>
<td>Change Font</td>
<td>Opens a standard font selection box. The options selected in the font dialog box are applied to the SmartLinker window content area.</td>
</tr>
<tr>
<td>Clear Log</td>
<td>Allows you to clear the SmartLinker window content area.</td>
</tr>
</tbody>
</table>
SmartLinker Configuration Window

The SmartLinker Configuration Window has three tabs. Each of the tabs is discussed in the following sections.

Editor Settings Tab

The Configuration Window Editor Settings Tab, as shown in Figure 1.4, has option buttons that let you select an editor type for SmartLinker, or for all Tools. Depending on the type of editor selected, the Editor Settings tab content changes.

Global Editor Option

In the view below, the Global Editor option has been selected.

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 of the file. Some Modifiers (editor options) can be specified in the editor command line. Once these options are stored, the behavior of the other tools that use the same entry changes when they are started the next time.
Figure 1.4 Editor Settings Tab - Global Editor
Local Editor Option

The Configuration Window Editor Settings tab with the Local Editor option selected is shown in Figure 1.5.

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

The 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 also changes when they are started the next time.

Figure 1.5 Editor Settings Tab - Local Editor
Editor started with Command Line Option

The Configuration Window Editor Settings tab with the Editor started with Command Line option selected is shown in Figure 1.6.

Figure 1.6 Editor Settings Tab - 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.

You enter the command that should be used to start the editor. The format for the editor command depends on the syntax that should be used to start the editor. Some Modifiers can be specified in the editor command line to refer to a line number of the named file.

Example:
For Winedit 32-bit versions, use (with an adapted path to the winedit.exe file):
C:\WinEdit32\WinEdit.exe %f /%l /%c

For Write.exe, use (with an adapted path to the write.exe file, note that write does not support line numbers):
C:\Winnt\System32\Write.exe %f
Editor Communication with DDE Option

The Configuration Window Editor Settings tab with the Editor Communication with DDE option selected is shown in Figure 1.7.

Figure 1.7 Editor Settings - Editor Communication with DDE

You must enter the Service and Topic Name as well as the Client Command to be used for a DDE connection to the editor. All entries can have modifiers for file name and line number as explained in the Modifiers section below.

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 was
detected.
- The %l modifier refers to the line number where the message was detected.

NOTE Be careful using the %l modifier. It can be used only with an editor which can
be started with a line number as a parameter. This is not the case for WinEdit
version 3.1 or lower or for Notepad. When you work with such an editor, you
can start it with the file name as a parameter and then select the menu entry ‘Go
to’ to jump to the line where the message was detected. In that case, the editor
command looks like: C:\WINAPPS\WINEDIT\Winedit.EXE %f
Please check your editor documentation to determine the command line which
should be used to start the editor.
Configuration Window Save Configuration Tab

All of the options for the Save operation are contained on the Save Configuration Tab of the Configuration Window, as shown in Figure 1.8.

Figure 1.8 Save Configuration Tab

In the Save Configuration tab, use the four checkboxes to choose which items to save to a project file when you save the configuration.

- **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 to the option and message settings are not saved and 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.
SmartLinker User Interface
SmartLinker Main Window

- **Appearance:** This item is related to many parts such as 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.

- **Environment Variables:** This item is related to the environment variable settings on the Environment tab. If this checkbox is set, the settings specified 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 no longer modify the options.

- **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 other parts of the configuration have changed. No confirmation will appear in any case when closing the linker.

**NOTE**  Most settings are stored only in the project configuration file. 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.
Configuration Window Environment Tab

All of the options for configuring environment variables are contained on the Environment Tab of the Configuration Window, as shown in the following figure:

Figure 1.9 Environment Tab

In the Environment tab, environment variables can be defined for the SmartLinker. Click the Add button to add new entries, the Change button to change an existing entry, and the Up and Down button to change the order of the entries.
Options Settings Window

The five tabs of the Options Settings Window, shown in Figure 1.10, allow you to set or reset SmartLinker options.

Figure 1.10 Option Settings Window

In addition to the Optimization tab, a tab is provided for each of the four option groups. Table 1.5 describes these four tabs.
A SmartLinker option is set when its checkbox is checked. To obtain a more detailed explanation about a specific option, select the option and then press the key F1 or the help button. To select an option, click once on the option text. The option text is then highlighted.

When the window is opened, no options are selected. Pressing the F1 key or the help button then shows the help for this window.

### Table 1.5  Option Settings Group Description

<table>
<thead>
<tr>
<th>Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Lists options related to the output files generation (what kind of files are to be generated).</td>
</tr>
<tr>
<td>Input</td>
<td>Lists options related to the input files.</td>
</tr>
<tr>
<td>Messages</td>
<td>Lists options controlling the generation of error messages.</td>
</tr>
<tr>
<td>Host</td>
<td>Lists host-specific options.</td>
</tr>
</tbody>
</table>
Message Settings Window

The Message Settings Window, shown in Figure 1.11, allows you to map messages to a different message class.

Depending on the message class, messages are shown in different colors 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 in both the manual and online help.

A tab is available for each error message class: Disabled, Information, Error, Warning and Fatal. Highlighting an entry on the Error tab, then clicking on the Information Command Button maps the error message to the information class.

Figure 1.11 Message Settings Window
Table 1.6 describes the message classes available in the Message Settings dialog box.

### Table 1.6 Message Class Description

<table>
<thead>
<tr>
<th>Message Class</th>
<th>Description</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disabled</td>
<td>Lists all disabled messages. Messages displayed in the list box will not be displayed by the SmartLinker.</td>
<td>None.</td>
</tr>
<tr>
<td>Information</td>
<td>Lists all information messages. Information messages inform about action taken by the SmartLinker.</td>
<td>Green</td>
</tr>
<tr>
<td>Warning</td>
<td>Lists all warning messages. When such a message is generated, linking of the input file continues and an absolute file is generated.</td>
<td>Blue</td>
</tr>
<tr>
<td>Error</td>
<td>Lists all error messages. When such a message is generated, linking of the input file continues but no absolute file is generated.</td>
<td>Red</td>
</tr>
<tr>
<td>Fatal</td>
<td>Lists all fatal error messages. When such a message is generated, linking of the input file stops immediately. Fatal messages cannot be changed. There are only listed to call context help.</td>
<td>Dark Red</td>
</tr>
</tbody>
</table>
Changing the Message Class

You can configure your own mapping of messages in the different classes using 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, 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 warning message ‘L1201: No stack defined’ as an error message:

1. Click the **Warning** tab to display the list of all warning messages.
2. Click on the string ‘L1201: No stack defined’ in the list box to select the message.
3. 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 active only 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 modifications you have made 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 the ‘Default’ button. To reset all messages to the default, click the ‘Reset All’ button.
About Dialog Box

The About dialog box, shown in Figure 1.12, can be opened with the Help->About command.

The About box contains extensive information. The main linker version is displayed separately on top of the dialog and the current directory and the versions of subparts of the linker are shown.

In addition, the About box contains all information needed to create a permanent license. The contents of the About box can be copied and pasted using standard Windows commands.

Click OK to close the dialog box.

During a linking session, the versions of linker subparts cannot be requested. They are displayed only when the linker currently is not processing.

Figure 1.12 The About Dialog Box
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, help for the first is shown. When no message is selected, pressing the F1 key 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 dialog box configuration settings and the options specified on the command line.

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

Using 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 Link button in the tool bar is selected or the enter key is pressed.

Linking a Previously Linked File

Previously executed commands can be displayed using the arrow on the right side of the command line. Select a command by clicking on it. It appears in the command line. The specified file will be linked as soon as the Link button 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 displayed with the list of all the prm files in the project directory. You can browse to get the name of the file you want 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 file 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:

```
>>in <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	stpla8.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 for different message formats.

Use SmartLinker Window Information

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

Use your 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 window, Editor Settings tab. The way error feedback is performed varies, depending on whether or not the editor can be started with a line number.
Line Number Specified on Command Line
An 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 occurred will be automatically opened, and the cursor will be placed on the line where the error was detected.

Line Number Cannot Be Specified on Command Line
An editor like WinEdit V31 or lower, Notepad, or 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 and 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 by pressing CTRL + V.
- Click Forward to jump to the position where the error was detected.
Environment Variables

This chapter describes the environment variables used by the SmartLinker. Other tools also use some of the same environment variables. For example, the Macro Assembler, and the Compiler use some of the environment variables. Refer to the respective tool manual for more information.

Various parameters of the SmartLinker can be set in the environment using environment variables. The syntax is always the same:

`VARIABLENAME "=" Definition`

**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: Environment File Specification”.

**NOTE** The project directory shown above can be set via the system environment variable “DEFAULTDIR: Default Current Directory”.

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: Environment File Specification”. If no definition can be found, a default value is assumed.

**NOTE** The environment can also be changed using the `-Env` SmartLinker option.
The Current Directory

The most important environment variable 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 is determined by the operating system or by the program that launches another program (for example, WinEdit).

For the UNIX operating system the directory in which an executable is started is also the current directory from which the binary file was 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 a desktop icon, the current directory is the working directory specified and associated with the icon.
- If the tool is launched by dragging a file onto the desktop icon, the desktop is the current directory.
- If the tool is launched by another 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 also changes 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: Default Current Directory” can be used.

The current directory is displayed, along with other information, using 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 file. 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 directory, the current file located in the same directory as 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

Path

Arguments:

Last installation path

Description:

When you install a tool, the installation script stores the installation destination directory in this variable.

Example:

Path=c:\install
Environment Variables
Global Initialization File (MCUTOOLS.INI - PC Only)

---

Group

**Arguments:**
Last installation program group.

**Description:**
When you install a tool, the installation script stores the created program group in this variable.

**Example:**
Group=ANSI-C Compiler

[Options] Section

---

DefaultDir

**Arguments:**
Default Directory to use.

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

**Example:**
DefaultDir=c:\install\project
Environment Variables
Global Initialization File (MCUTOOLS.INI - PC Only)

[LINKER] Section

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 case.

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.

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.
SaveOptions

Arguments:

1/0

Description:

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

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.

TipFilePos

Arguments:

Any integer, e.g. 236

Description:

Index of the tip which is actually shown - used to display different tip every time.
ShowTipOfDay

**Arguments:**

0/1

**Description:**
Decides if the Tip of the Day dialog should be shown at startup.
1: show Tip of the Day at startup,
0: show Tip of the Day only when opened from the help menu.

TipTimeStamp

**Arguments:**

Date

**Description:**
This entry is used to record the time that new tips became 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
```
Environment Variables
Global Initialization File (MCUTOOLS.INI - PC Only)

[Editor] Section

Editor_Name

Arguments:
The name of the global editor

Description:
Specifies the name displayed in the global editor. This entry has only a descriptive effect. Its content does not apply to starting the editor.

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

Editor_Exe

Arguments:
The name of the executable file of the global editor

Description:
Specifies the file name (including its 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 active only when this entry is present and not empty.

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

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 built 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 tab.

Example:

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

Local Configuration File (Usually project.ini)

MCUTOOLS.INI Example

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

```
[Installation]
Path=c:\Freescale
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 reads DEFAULT.ENV and does not change its content in any way. 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 linker can use any file name for the project configuration file, which has 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 will use the same entries. This is important mainly when options available in only one version are 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 can be used for both linker versions.

The current directory is always the directory where the configuration file is. If a configuration file in a different directory is loaded, then the current directory also changes. When the current directory changes, the DEFAULT.ENV file is reloaded. Whenever a configuration file is loaded or stored, the options in the environment variable “LINKOPTIONS: Default SmartLinker Options” are also reloaded and added to the project options. This behavior is important to note when different DEFAULT.ENV exist in different directories and contain incompatible LINKOPTIONS options. When a project
Environment Variables

Local Configuration File (Usually project.ini)

is loaded using the first DEFAULT.ENV, its LINKOPTIONS are added to the configuration file. If this configuration is then stored in a different directory where a DEFAULT.ENV exists with incompatible options, the linker adds options and reports the inconsistency. A message appears to report that the DEFAULT.ENV options were not added. If this occurs, you can either remove the option from the configuration file using the advanced option dialog or you can remove the option from the DEFAULT.ENV with the shell or a text editor, depending upon which options should be used in the future.

At startup there are two ways to load a configuration:

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

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

[Editor] Section

Editor_Name

Arguments:
The name of the local editor

Description:
Specifies the name displayed in the local editor. This entry has only a descriptive effect. Its content does not apply to starting the editor.

Saved:
Only with Editor Configuration set in the File > Configuration... > Save Configuration Tab. This entry has the same format as the global editor configuration in the mctools.ini file.
Environment Variables
Local Configuration File (Usually project.ini)

Editor_Exe

**Arguments:**
The name of the executable file of the local editor

**Description:**
Specifies the file name which is called for showing a text file when the local editor setting is active. In the editor configuration dialog, the local editor selection is active only when this entry is present and not empty.

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

Editor_Opts

**Arguments:**
The options to use the local editor

**Description:**
Specifies the 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 built 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 Tab. This entry has the same format as 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

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 Tab.

CurrentCommandLine

**Arguments:**
String with the command line. For example: fibo.prm -wl

**Description:**
The currently visible command line content.

**Saved:**
Only with Appearance set in the File > Configuration... > Save Configuration Tab.
Environment Variables
Local Configuration File (Usually project.ini)

---

**StatusBarEnabled**

**Arguments:**

1/0

**Special:**

This entry is only considered at startup. Later load operations do not use it.

**Description:**

Is the status bar currently enabled?

1: the status bar is visible.

0: the status bar is hidden.

**Saved:**

Only with Appearance set in the File > Configuration... > Save Configuration Tab.

---

**ToolbarEnabled**

**Arguments:**

1/0

**Special:**

This entry is only considered at startup. Later load operations do not use it.

**Description:**

Is the tool bar currently enabled?

1: the tool bar is visible.

0: the tool bar is hidden.

**Saved:**

Only with Appearance set in the File > Configuration... > Save Configuration Tab.
Environment Variables
Local Configuration File (Usually project.ini)

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.
Changes of this entry do not show the “*” in the title

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

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

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 Tab.

Example:
WindowFont=-16,500,0,Courier
Environment Variables
Local Configuration File (Usually project.ini)

Options

Arguments:
W2

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

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

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.

Saved:
Only with Editor Configuration set in the File > Configuration... > Save Configuration Tab.
Environment Variables
Local Configuration File (Usually project.ini)

---

**EditorCommandLine**

**Arguments:**

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

**Description:**

Command line content to open a file.

**Saved:**

Only with Editor Configuration set in the **File > Configuration... > Save Configuration** Tab.

---

**EditorDDEClientName**

**Arguments:**

Client command. For example, `“[open (%f)]”`

**Description:**

Name of the client for DDE editor configuration.

**Saved:**

Only with Editor Configuration set in the **File > Configuration... > Save Configuration** Tab.
**EditorDDETopicName**

**Arguments:**
Topic name. For example, "system"

**Description:**
Name of the topic for DDE editor configuration.

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

---

**EditorDDESERVICEName**

**Arguments:**
Service name. For example, "system"

**Description:**
Name of the service for DDE editor configuration.

**Saved:**
Only with Editor Configuration set in the **File > Configuration... > Save Configuration** Tab.
**Configuration File 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,-1,1390,107,1103,643
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\Freescale\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/.hidefaults) over different lines using the line continuation character ‘\’:

Example:
COMPOPTIONS=\n-W2 \n-Wpd

This is the same as:
COMPOPTIONS=-W2 -Wpd

However, this feature may be dangerous, when using it together with paths, for example,
GENPATH=.\nTEXTFILE=.	xt

Will result in:
GENPATH=\TEXTFILE=.	xt

To avoid such problems, we recommend that you use a semicolon ‘;’ at the end of a path if there is a ‘\’ at the end:
GENPATH=.;
TEXTFILE=.	xt
Environment Variable Details

The remainder of this section is devoted to describing each of the environment variables available for the SmartLinker. Table 2.1 contains options. Each option is divided into several sections.

Table 2.1 Environment Variable Description

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tools</td>
<td>Lists tools which use this variable.</td>
</tr>
<tr>
<td>Synonym</td>
<td>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.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Specifies the syntax of the option in a EBNF format.</td>
</tr>
<tr>
<td>Arguments</td>
<td>Describes and lists optional and required arguments for the variable.</td>
</tr>
<tr>
<td>Default</td>
<td>Shows the default setting for the variable or none.</td>
</tr>
<tr>
<td>Description</td>
<td>Provides a detailed description of the option and how to use it.</td>
</tr>
<tr>
<td>Example</td>
<td>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.</td>
</tr>
<tr>
<td>See also</td>
<td>Names related sections.</td>
</tr>
</tbody>
</table>
**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 Entry in Absolute File

Tools:
   Compiler, Assembler, SmartLinker, Libmaker

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 variables "USERNAME: User Name in Object File" and "INCLUDETIME: Creation Time in Object File"
Environment Variables
Environment Variable Details

DEFAULTDIR: Default Current Directory

Tools:
Compiler, Assembler, SmartLinker, Decoder, Debugger, Libmaker, 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).

Example:
DEFAULTDIR=C:\INSTALL\PROJECT

See also:
“The Current Directory” and “Global Initialization File (MCUTOOLS.INI - PC Only)”

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

Tools:
Compiler, SmartLinker, Decoder, Debugger, Libmaker, 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.

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

See also:
None

NOTE This is an environment variable on the system level (global environment variable). It cannot be specified in a default environment file (DEFAULT.ENV/.hidefaults).
**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 a source file `test.prm`, an error list file `dir1\test.err` will be generated.

```plaintext
ERRORFILE=%p\errors.txt
```

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, or 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.

---

**GENPATH: Define Paths to Search for Input Files**

**Tools:**

- Compiler, Assembler, SmartLinker, Decoder, Debugger

**Synonym:**

- HIPATH

**Syntax:**

```plaintext
"GENPATH= " {<path>}
```

**Arguments:**

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

**Description:**

The SmartLinker looks 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`.

**Example:**

```plaintext
GENPATH=\obj;..\.\lib;
```
Environment Variables
Environment Variable Details

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.

INCLUDETIME: Creation Time in Object File

Tools:
  Compiler, Assembler, SmartLinker, Libmaker

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 undesirable 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
LINKOPTIONS: Default SmartLinker Options

Tools:

SmartLinker

Synonym:

None

Syntax:

"LINKOPTIONS=" {<option>}

Arguments:

<option>: SmartLinker command line option.

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 that you do not have to specify them each time a file is linked.

Example:

LINKOPTIONS=-W2

See also:

SmartLinker Options Chapter

OBJPATH: Object File Path

Tools:

Compiler, Assembler, SmartLinker, Decoder, Debugger

Synonym:

None

Syntax:

"OBJPATH=" {<path>}

HC(S)08/RS08 and HC(S)12 Build Tools Utilities Manual
Environment Variables
Environment Variable Details

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

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.

Default:
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
SRECORD: S Record File Format

Tools:
Assembler, SmartLinker, Burner

Synonym:
None

Syntax:
"SRECORD=" <RecordType>

Arguments:
<Record Type>: Force the type for the 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 generates a Freescale 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).

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, an S1 record is generated. If the address is coded on 3 bytes, an S2 record is generated. Otherwise an S3 record is generated.

Example:
SRECORD=S2

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

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 stores the map file it produces in the first directory specified there. If TEXTPATH is not set, the generated map file is stored in the directory where the prm file was found.

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

Tools:
Compiler, Assembler, SmartLinker, Debugger, Libmaker

Synonym:
None

Syntax:
“TMP=“ <directory>

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

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”.

Example:
TMP=C:\TEMP

See also:
“The Current Directory”

NOTE This is an environment variable on the system level (global environment variable). It cannot be specified in a default environment file (DEFAULT.ENV/.hidefaults).
USERNAME: User Name in Object File

Tools:

Compiler, Assembler, SmartLinker, Libmaker

Synonym:

None

Syntax:

"USERNAME= " <user>

Arguments:

<user>: Name of user.

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:

“COPYRIGHT: Copyright Entry in Absolute File” and “INCLUDETIME: Creation Time in Object File”
SmartLinker Files

This chapter describes the input and output files used by the SmartLinker.

- “Input Files”
- “Output 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: Define Paths to Search for Input Files”. 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 -Add: Additional Object/Library File option.

The linker looks for the object files first in the project directory, then in the directories enumerated in “OBJPATH: Object File Path” and finally in the directories enumerated in “GENPATH: Define Paths to Search for Input Files”. The binary files must be valid Freescale (former 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: Absolute Path”. 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.

S Record Files
After successful linking session and if the -B: Generate S-Record file option is present, the SmartLinker generates an 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 S Record file depends on the setting from the variable “SRECORD: S Record File Format”.

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

This file is written to the directory given in the environment variable “ABSPATH: Absolute Path”. 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: Text Path”. 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, etc.).

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.

Examples:

```c
int bar;
void fun(void) {
    bar = 0;
}
```

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

```c
void fun(void) {
    fun();
}
```

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:

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

In the above example, the compiler could optimize this to:

```c
int fun(void) {
    bar fun:Label:
    ...
    fun_Label:
    if (bar == 3) bar = 0;
    return;
```
Here the compiler will generate a local branch inside fun to a local subroutine. This produces a relocation/fixup into fun, that is for the linker fun 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: Error File Name Specification").

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 EDOM with 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.
Figure 3.1 Error File Creation

SmartLinker Files
Output Files

![Diagram showing the creation of error files with SmartLinker]

- `.prm` 1. current dir 2. GENPATH
- `.sx` 1. ABSPATH 2. Source file path
- `.abs` 1. current dir 2. OBJPATH 3. GENPATH
- `.map` 1. TEXTPATH 2. Source file path
- `ERRORFILE` 1. ERR.TXT 2. EDOUT

 realised
SmartLinker Files

Output Files
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 the name of a parameter file to be linked. SmartLinker options may be specified on the command line or in the LINKOPTIONS environment 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 the LINKOPTIONS 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 options available for the SmartLinker. **Table 4.1** lists the details described for each of the options.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td>Specifies what sort of influence this option has.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Specifies the syntax of the option in a EBNF format.</td>
</tr>
<tr>
<td>Arguments</td>
<td>Describes and lists optional and required arguments for the option.</td>
</tr>
<tr>
<td>Default</td>
<td>(Where used): Shows the default setting for the option.</td>
</tr>
<tr>
<td></td>
<td>(Where not used): No default setting for the option.</td>
</tr>
<tr>
<td>Description</td>
<td>Provides a detailed description of the option and how to use it.</td>
</tr>
<tr>
<td>Example</td>
<td>Gives an example of usage, and effects of the option where possible.</td>
</tr>
<tr>
<td></td>
<td>SmartLinker settings, source code and/or SmartLinker prm files are displayed</td>
</tr>
<tr>
<td></td>
<td>where applicable. The examples shows an entry in the default.env for PC or</td>
</tr>
<tr>
<td></td>
<td>in the .hidefaults for UNIX.</td>
</tr>
<tr>
<td>See also</td>
<td>(Where used): Names related options.</td>
</tr>
</tbody>
</table>

### -Add: Additional Object/Library File

**Group:**

INPUT

**Syntax:**

```
"-Add" <FileList>
```

**Arguments:**

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

**Description:**

With the this option, 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.

**Example:**

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"}
```

```
inker.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:**

[NAMES: List Files Building the Application](#)

**NOTE**

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

---

**-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 always takes 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 Freescale (former Hiware) format. In the Freescale 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-"  "CCDDEEEFFFFF"
AllocChange:"CCC"  "AAAAABBD"  "EEEEFFFFF---"
AllocFirst:  "BBD"  "AAAACC"  "EEEEFFFFF---"
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.

**-AsROMLib: Link as ROM Library**

**Group:**
OUTPUT

**Syntax:**
```
"-AsROMLib"
```

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

**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
```

**-B: Generate S-Record file**

**Group:**
OUTPUT

**Syntax:**
```
"-B"
```

**Arguments:**
```
none
```
SmartLinker Options

SmartLinker Option Details

Default:
  Disabled

Description:
This option specifies that in addition to an absolute file, also an 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

-CAllocUnusedOverlap: Allocate Not Referenced Overlap Variables
(Freescale)

Group:
  OPTIMIZATION

Syntax:
  “-CAllocUnusedOverlap”

Arguments:
  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
```

---

**-Ci: Link Case Insensitive**

**Group:**

**INPUT**

**Syntax:**

```
"-Ci"
```

**Arguments:**

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 severe 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:**

```c
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 an “implicit parameter declaration” warning.
SmartLinker Options
SmartLinker Option Details

-Cocc: Optimize Common Code (ELF)

Group:
  OPTIMIZATION

Syntax:
  "-Cocc" ["=" ["D"] ["C"]]

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

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”.

-CRam: Allocate Non-specified Constant Segments in RAM (ELF)

Group:
  OPTIMIZATION

Syntax:
  "-CRam"

Arguments:
  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.

-Dist: Enable Distribution Optimization (ELF)

Group:
OPTIMIZATIONS

Syntax:
"-Dist"

Arguments:
none

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

-DistFile: Specify Distribution File Name (ELF)

Group:
OPTIMIZATIONS

Syntax:
"-DistFile"<file name>
SmartLinker Options
SmartLinker Option Details

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, all distributed functions and how the compiler has to reallocate them are listed.

Example:

LINKOPTIONS=-DistFileMyFile

-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
-DistOpti: Choose Optimizing Method (ELF)

Group: OPTIMIZATIONS

Syntax:

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

"CodeSize": Priority is to minimize the code size.

Arguments:

"FillBanks": Priority is to fill the banks.

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

-DistSeg: Specify Distribution Segment Name (ELF)

OPTIMIZATIONS

Syntax:

"-DistSeg"<segment name>

Arguments:

<segment name>: Name of the distribution segment.
**SmartLinker Options**

**SmartLinker Option Details**

---

**Default:**

DISTRIBUTE

**Description:**

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

**Example:**

```
LINKOPTIONS=-DistSegMyDistributionSegment
```

---

**-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.

**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:

```
INIT entry
```

in the prm file.
-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.

Description:

This option sets an environment variable.

Example:

"-EnvOBJPATH=\sources\obj"

This is the same as:

OBJPATH=\sources\obj

in the default.env

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

Group:

INPUT

Syntax:

"-F" ("A" | "E" | "H" | "6")

Arguments:

none

Default:

"-FA"
SmartLinker Options

SmartLinker Option Details

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 `-FE`, this automatism can be overridden and only ELF files are correctly recognized.
- With `-FH` only Freescale (former Hiware) files are known.
- With `-F6` set, the linker produces a V2.6 Freescale (former Hiware) absolute file.

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

-H: Prints the List of All Available Options

Group:

OUTPUT

Syntax:

```
-H
```

Arguments:

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.
Linker option output ot -H:
- `F` Object File Format
- `Fh` Freescale
- `FEo` Compatible ELF (DWARF 1.1/DWARF 2.0)
- `Fa` Automatic Detection
- `F6` Freescale V2.6
-L: Add a Path to Search Path (ELF)

Group:
INPUT

Syntax:
"-L" <Directory>

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

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:\freescale\obj

See also:
OBJPATH: Object File Path

-Lic: Print License Information

Group:
Various

Syntax:
"-Lic"

Arguments:
none
SmartLinker Options
SmartLinker Option Details

Description:
This option 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:

-Lic

-LicA: License Information About Every Feature in Directory

Group:
Various

Syntax:
"-LicA

Arguments:
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:

-LicA

-LicBorrow: Borrow License Feature

Group:
Host

Syntax:
"-LicBorrow"<feature>[:";<version>]":<Date>
Arguments:
<feature>: the feature name to be borrowed (e.g. HI100100).
.VERSION>: optional version of the feature to be borrowed (e.g. 3.000).
<date>: date with optional time until when the feature shall be borrowed (e.g. 15-Mar-2004:18:35).

Description:
This option allows to borrow a license feature until a given date/time. Borrowing allows you to use a floating license even if disconnected from the floating license server.

You need to specify the feature name and the date until you want to borrow the feature. If the feature you want to borrow is a feature belonging to the tool where you use this option, then you do not need to specify the version of the feature (because the tool knows the version). However, if you want to borrow any feature, you need to specify as well the feature version of it. You can check the status of currently borrowed features in the tool about box.

You only can borrow features, if you have a floating license and if your floating license is enabled for borrowing. See as well the provided FLEXlm documentation about details on borrowing.

Example:
-LicBorrowHI100100;3.000:12-Mar-2004:18:25

-LicWait: Wait Until Floating License Is Available from Floating License Server

Group:
Host

Syntax:
"-LicWait"

Arguments:
none
**SmartLinker Options**

*SmartLinker Option Details*

---

**Description:**
By default, if a license is not available from the floating license server, then the application will immediately return. With `-LicWait` set, the application will wait (blocking) until a license is available from the floating license server.

**Example:**
- `-LicWait`

---

**-M: Generate Map File**

**Group:**
- OUTPUT

**Syntax:**
- `"-M"`

**Arguments:**
- none

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

**Example:**
```bash
LINKOPTIONS=-M
```
This is the same as using the command:
```bash
MAPFILE ALL
```
in the prm file.

**See also:**
- [MAPFILE: Configure Map File Content](#)
-N: Display Notify Box

Group:
MESSAGE

Syntax:
“-N”

Arguments:
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.

-NoBeep: No Beep in Case of an Error

Group:
MESSAGE

Syntax:
“-NoBeep”

Arguments:
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 off using this option.
SmartLinker Options
SmartLinker Option Details

Example:
none

-NoEnv: Do Not Use Environment

Group:
Startup. (This option cannot be specified interactively.)

Syntax:
"-NoEnv"

Arguments:
none

Description:
This option can be specified only at the command line while starting the application. It cannot be specified in any other circumstance, including the default.env file, the command line, etc.

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:
Environment Variables

-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.

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:

LINK: Specify Name of Output File

-OCopy: Optimize Copy Down (ELF)

Group:

OPTIMIZATION

Syntax:

“-OCopy” (“On” | “Off”)

Arguments:

On: Do the optimization.

Off: Optimization disabled.

Default:

-OCopyOn.
SmartLinker Options
SmartLinker Option Details

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
Freescale (former Hiware) format cannot be switched off.

Example:

```
LINKOPTIONS=-OCopyOn
```

-Prod: Specify Project File at Startup (PC)

Group:
none. (This option cannot be specified interactively.)

Syntax:
```
"-Prod"<file>
```

Arguments:

<file>: Name of a project or project directory.

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

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

Example:

```
linker.exe -prod=project.ini
```
**SmartLinker Options**

**SmartLinker Option Details**

---

**-S: Do Not Generate DWARF Information (ELF)**

*Group:*  
OUTPUT

*Syntax:*  
```
-S
```

*Arguments:*  
none

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

*Example:*  
```
LINKOPTIONS=-S
```

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

---

**-SFixups: Creating Fixups (ELF)**

*Group:*  
OUTPUT

*Syntax:*  
```
-SFixups
```

*Arguments:*  
none

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
```

---

**-StatF: Specify Name of Statistic File**

**Group:**

OUTPUT

**Syntax:**

```
-StatF=\"<fileName>\"
```

**Arguments:**

<fileName>: Name for the file to be written.

**Description:**

With this option set, the linker generates a statistic file. In this file, each allocated object is reported with its 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
```

---

**-V: Prints SmartLinker Version**

**Group:**

OUTPUT

**Syntax:**

```
-\V
```

---
Arguments:
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

-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.
SmartLinker Options
SmartLinker Option Details

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: Display Notify Box, a dialog box is still possible.

Example:
-ViewHidden fibo.prm

-W1: No Information Messages

Group:
MESSAGE

Syntax:
“-W1”

Arguments:
none

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

Example:
LINKOPTIONS=-W1

-W2: No Information and Warning Messages

Group:
MESSAGE

Syntax:
“-W2”
Arguments:
none

Description:
Suppresses all messages of type INFORMATION and WARNING, only ERRORs are printed.

Example:
LINKOPTIONS=-W2

-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:

-WStdout: Write to Standard Output
-WOutFile: Create Error Listing File

-\textbf{\texttt{Wmsg8x3}: Cut File Names in Microsoft Format to 8.3 (PC)}

\textbf{Group:}

MESSAGE

\textbf{Syntax:}

```
-\texttt{Wmsg8x3}
```

\textbf{Arguments:}

none

\textbf{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 eight characters with not more than a three 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.

\textbf{Example:}

```
x:\textbackslash\mysourcefile.prm\{3\}: \texttt{INFORMATION} C2901: Unrolling loop
```

With the option \texttt{-Wmsg8x3} set, the above message will be:

```
x:\textbackslash\mysource.c\{3\}: \texttt{INFORMATION} C2901: Unrolling loop
```
-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)

**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 may be specified in decimal. Hexadecimal needs a 0X prefix, (for gray errors: -WmsgCE0x808080).

**Example:**
-WmsgCE255 changes the error messages to blue.

-WmsgCF: RGB Color for Fatal Messages

**Group:**
MESSAGE

**Scope:**
Function

**Syntax:**
"-WmsgCF" <RGB>
SmartLinker Options
SmartLinker Option Details

Arguments:

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

Default:

-WmsgCF8388608 (r80 g00 b00, dark red)

Description:
With this option it is possible to change the fatal message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and may be specified in decimal. Hexadecimal needs a 0X prefix, (for gray fatal errors: -WmsgCF0x808080).

Example:

-WmsgCF255 changes the fatal messages to blue.

-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)

Description:
With this option it is possible to change the information message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and may be specified in decimal. Hexadecimal needs a 0X prefix, (for gray messages: -WmsgCl0x808080).
Example:

-\WmsgCI255 changes the information messages to blue.

-\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)

Description:

With this option it is possible to change the user message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and may be specified in decimal. Hexadecimal needs a 0X prefix, (for gray messages: -WmsgCU0x808080).

Example:

-\WmsgCU255 changes the user messages to blue.

-\WmsgCW: RGB Color for Warning Messages

Group:

MESSAGE

Scope:

Function
**SmartLinker Options**

**SmartLinker Option Details**

---

**Syntax:**

```
"-WmsgCW" <RGB>
```

**Arguments:**

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

**Default:**

`-WmsgCW255` (r00 g00 bFF, blue)

**Description:**

With this option it is possible to change the warning message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and may be specified in decimal. Hexadecimal needs a 0X prefix, (for gray warnings: `-WmsgCW0x808080`).

**Example:**

`-WmsgCW0` changes the warning messages to black.

---

**-WmsgFb (-WmsgFbv, -WmsgFbm): 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 contains only 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 the 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
```
-WmsgFi (-WmsgFiv, -WmsgFim): 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:

```
PLACEMENT
  .text INTO READONLY 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
```

Setting the format to Microsoft, less information is displayed:

```
LINKOPTIONS=-WmsgFm
```

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

See also:

- `-WmsgFb (-WmsgFvb -WmsgFbm): Set Message File Format for Batch Mode`

---

**-WmsgFob: Message Format for Batch Mode**

**Group:**

MESSAGE

**Syntax:**

```
"-WmsgFob" <string>
```

**Arguments:**

<string>: format string ("WmsgFob-Supported Format String Symbols").

**Default:**

```
-WmsgFob"%*f%f&x"(%l): %K %d: %m\n"
```
**SmartLinker Options**

**SmartLinker Option Details**

**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:\freescale\sourcefile.prmx)

**Example:**
```
LINKOPTIONS=-WmsgFob"%'%f%e%'(%l): %k %d: %m\n"
```
Produces a message in following format:

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

**See also:**
- Environment variable: ERRORFILE: Error File Name Specification
- Option -WmsgFb (-WmsgFbv, -WmsgFbm): Set Message File Format for Batch Mode
- Option -WmsgFi (-WmsgFiv, -WmsgFim): Set Message File Format for Interactive Mode
- Option -WmsgFonp: Message Format for No Position Information
- Option -WmsgFonf: Message Format for no File Information
- Option --WmsgFoi: Message Format for Interactive Mode

**Table 4.2 WmsgFob-Supported Format String Symbols**

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%s</td>
<td>Source Extract</td>
<td></td>
</tr>
<tr>
<td>%p</td>
<td>Path</td>
<td>x:\freescale\</td>
</tr>
<tr>
<td>%f</td>
<td>Path and name</td>
<td>x:\freescale\sourcefile</td>
</tr>
<tr>
<td>%n</td>
<td>File name</td>
<td>sourcefile</td>
</tr>
<tr>
<td>%e</td>
<td>Extension</td>
<td>.prmx</td>
</tr>
<tr>
<td>%N</td>
<td>File (8 chars)</td>
<td>sourcefi</td>
</tr>
<tr>
<td>%E</td>
<td>Extension (3 chars)</td>
<td>.prm</td>
</tr>
<tr>
<td>%l</td>
<td>Line</td>
<td>3</td>
</tr>
<tr>
<td>%c</td>
<td>Column</td>
<td>47</td>
</tr>
</tbody>
</table>
-WmsgFoi: Message Format for Interactive Mode

**Group:**

MESSAGE

**Syntax:**

"-WmsgFoi"<string>

**Arguments:**

<string>: format string ("WmsgFoi-Supported Format String Symbols").

**Default:**

-<WmsgFoi>

**Description:**

With this option it is possible to modify the default message format in interactive mode. If the source file is x\freescale\sourcefile, the following formats are supported: ("WmsgFoi-Supported Format String Symbols")

### Table 4.2  WmsgFob-Supported Format String Symbols (continued)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%o</td>
<td>Pos</td>
<td>1000</td>
</tr>
<tr>
<td>%K</td>
<td>Uppercase kind</td>
<td>ERROR</td>
</tr>
<tr>
<td>%k</td>
<td>Lowercase kind</td>
<td>error</td>
</tr>
<tr>
<td>%d</td>
<td>Number</td>
<td>L1051</td>
</tr>
<tr>
<td>%m</td>
<td>Message</td>
<td>text</td>
</tr>
<tr>
<td>%&quot;</td>
<td>&quot; if full name contains a space</td>
<td>&quot;</td>
</tr>
<tr>
<td>%'</td>
<td>' if full name contains a space</td>
<td></td>
</tr>
<tr>
<td>%%</td>
<td>Percent</td>
<td>%</td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
<td></td>
</tr>
</tbody>
</table>

%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=-WmsgFoi"%f%f (%l): %k %d: %m\n"
```

Produces a message in following format:

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

See also:

- `Environment variable: ERRORFILE: Error File Name Specification`
- `Option `-WmsgFb (-WmsgFbv, `-WmsgFbm): Set Message File Format for Batch Mode`
- `Option `-WmsgFi (-WmsgFiv, `-WmsgFim): Set Message File Format for Interactive Mode`
- `Option `-WmsgFonp: Message Format for No Position Information`
- `Option `-WmsgFonf: Message Format for no File Information`
- `Option `-WmsgFob: Message Format for Batch Mode`

### Table 4.3 WmsgFoi-Supported Format String Symbols

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%s</td>
<td>Source Extract</td>
<td></td>
</tr>
<tr>
<td>%p</td>
<td>Path</td>
<td>x:\freescale\</td>
</tr>
<tr>
<td>%f</td>
<td>Path and name</td>
<td>x:\freescale\sourcefile</td>
</tr>
<tr>
<td>%n</td>
<td>File name</td>
<td>sourcefile</td>
</tr>
<tr>
<td>%e</td>
<td>Extension</td>
<td>.prmx</td>
</tr>
<tr>
<td>%N</td>
<td>File (8 chars)</td>
<td>sourcefi</td>
</tr>
<tr>
<td>%E</td>
<td>Extension (3 chars)</td>
<td>.prm</td>
</tr>
<tr>
<td>%l</td>
<td>Line</td>
<td>3</td>
</tr>
<tr>
<td>%c</td>
<td>Column</td>
<td>47</td>
</tr>
<tr>
<td>%o</td>
<td>Pos</td>
<td>1234</td>
</tr>
<tr>
<td>%K</td>
<td>Uppercase kind</td>
<td>ERROR</td>
</tr>
<tr>
<td>%k</td>
<td>Lowercase kind</td>
<td>error</td>
</tr>
<tr>
<td>%d</td>
<td>Number</td>
<td>L1000</td>
</tr>
</tbody>
</table>
### -WmsgFonf: Message Format for no File Information

**Group:**

MESSAGE

**Syntax:**

```
"-WmsgFonf"<string>
```

**Arguments:**

- `<string>`: format string ("WmsgFonf-Supported String Format Symbols").

**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.

**Example:**

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

Produces a message in following format:

```
information L10324: Linking successful
```
**SmartLinker Options**

*SmartLinker Option Details*

---

### Table 4.4 WmsgFonf-Supported String Format Symbols

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%K</td>
<td>Uppercase kind</td>
<td>ERROR</td>
</tr>
<tr>
<td>%k</td>
<td>Lowercase kind</td>
<td>error</td>
</tr>
<tr>
<td>%d</td>
<td>Number</td>
<td>L10324</td>
</tr>
<tr>
<td>%m</td>
<td>Message</td>
<td>text</td>
</tr>
<tr>
<td>%%</td>
<td>Percent</td>
<td>%</td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
<td>\n</td>
</tr>
</tbody>
</table>

---

**-WmsgFonp: Message Format for No Position Information**

**Group:**

MESSAGE

**Syntax:**

```
"-WmsgFonp"<string>
```

**Arguments:**

`<string>`: format string (See: "WmsgFonp-Supported String Formats").

**Default:**

```
-WmsgFonp"%%f&e\n": %K %d: %m\n"
```

**Description:**

Sometimes there is no position information available for a message (e.g. if a message is not related to a certain position). Then this message format string is used. Supposing that the source file is `x\freescale\sourcefile.prmx`, the following formats are supported ("WmsgFonp-Supported String Formats").

**Example:**

```
LINKOPTIONS=-WmsgFonf"%k %d: %m\n"
```
Produces a message in following format:

```
information L10324: Linking successful
```

See also:
- Environment variable: ERRORFILE: Error File Name Specification
- Option -WmsgFb (-WmsgFbv, -WmsgFbm): Set Message File Format for Batch Mode
- Option -WmsgFi (-WmsgFiv, -WmsgFim): Set Message File Format for Interactive Mode
- Option -WmsgFonp: Message Format for No Position Information
- Option -WmsgFoi: Message Format for Interactive Mode
- Option -WmsgFob: Message Format for Batch Mode

### Table 4.5 WmsgFonp-Supported String Formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%p</td>
<td>Path</td>
<td>x:\freescale\</td>
</tr>
<tr>
<td>%f</td>
<td>Path and name</td>
<td>x:\freescale\sourcefile</td>
</tr>
<tr>
<td>%n</td>
<td>File name</td>
<td>sourcefile</td>
</tr>
<tr>
<td>%e</td>
<td>Extension</td>
<td>.prmx</td>
</tr>
<tr>
<td>%N</td>
<td>File (8 chars)</td>
<td>sourcefi</td>
</tr>
<tr>
<td>%E</td>
<td>Extension (3 chars)</td>
<td>.prm</td>
</tr>
<tr>
<td>%K</td>
<td>Uppercase kind</td>
<td>ERROR</td>
</tr>
<tr>
<td>%k</td>
<td>Lowercase kind</td>
<td>error</td>
</tr>
<tr>
<td>%d</td>
<td>Number</td>
<td>L10324</td>
</tr>
<tr>
<td>%m</td>
<td>Message</td>
<td>text</td>
</tr>
<tr>
<td>%&quot;</td>
<td>“ if full name contains a space</td>
<td>“</td>
</tr>
<tr>
<td>%’</td>
<td>’ if full name contains a space</td>
<td></td>
</tr>
<tr>
<td>%%</td>
<td>Percent</td>
<td>%</td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
<td></td>
</tr>
</tbody>
</table>
-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: Number of Information Messages
Option -WmsgNw: Number of Warning Messages

-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 10 information messages are logged.

See also:

Option -WmsgNi: Number of Information Messages
Option -WmsgNw: Number of Warning Messages
Option -WmsgNe: Number of Error Messages

-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.
**SmartLinker Options**

**SmartLinker Option Details**

**Description:**
The application produces some messages which are not in the normal message categories (WARNING, INFORMATION, WRORR, 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.

**Example:**

```
-WmsgNu=c
```

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

---

**-WmsgNw: Number of Warning Messages**

**Group:**

MESSAGE

**Syntax:**

```
-WmsgNw <number>
```

**Arguments:**

<number>: Maximum number of warning messages.

**Default:**

50
**Description:**
With this option the number of warning messages can be set.

**Example:**

```
LINKOPTIONS=-WmsgNw15
```

Only 15 warning messages are logged.

**See also:**
- Option `-WmsgNi`: Number of Information Messages
- Option `-WmsgNw`: Number of Warning Messages
- Option `-WmsgNe`: Number of Error Messages

---

**-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.
-WmsgSe: Setting a Message to Error

**Group:**
MESSAGE

**Syntax:**
```
-WmsgSe  <number>
```

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

**Description:**
Allows changing a message to an error message.

**Example:**
```
LINKOPTIONS=-WmsgSe1201
```

**See also:**
Option -WmsgSi: Setting a Message to Information
Option -WmsgSw: Setting a Message to Warning
Option -WmsgSd: Setting a Message to Disable
-WmsgSi: Setting a Message to Information

Group:
MESSAGE

Syntax:
"-WmsgSi" <number>

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

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

Example:
LINKOPTIONS=-WmsgSi1201

See also:
Option -WmsgSd: Setting a Message to Disable
Option -WmsgSw: Setting a Message to Warning
Option -WmsgSe: Setting a Message to Error

-WmsgSw: Setting a Message to Warning

Group:
MESSAGE

Syntax:
"-WmsgSw" <number>

Arguments:
<number>: Error number to be a warning, for example, 1201.
SmartLinker Options
SmartLinker Option Details

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

Example:

   LINKOPTIONS=-WmsgSw1201

See also:
Option -WmsgSi: Setting a Message to Information
Option -WmsgSd: Setting a Message to Disable
Option -WmsgSe: Setting a Message to Error

-WOutFile: Create Error Listing File

Group:
MESSAGE

Syntax:

   "-WOutFile" ("On" | "Off")

Arguments:
none

Default:
Error listing file is created.

Description:
This option controls whether or not an 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: Error File Name Specification.
Example:

-\WOutFileOn
  The error file is created as specified with ERRORFILE.
-\WOutFileOff
  No error file is created.

See also:

Option -WErrFile: Create “err.log” Error File
Option -WStdout: Write to Standard Output

-WStdout: Write to Standard Output

Group:
MESSAGE

Syntax:

  “-WStdout” (“On” | "Off")

Arguments:

  none

Default:

  Output is written to stdout.

Description:

In Windows applications, the usual standard streams are available, but text written to them
does not appear anywhere unless explicitly requested by the calling application. With this
option controls whether 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: Create “err.log” Error File
Option -WOutFile: Create Error Listing File
Linking Issues

Object Allocation

The whole object allocation is performed through The SEGMENTS Block (ELF) or Segments and PLACEMENT Block.

The SEGMENTS Block (ELF)
The SEGMENTS Block is optional. It only increases the readability of the linker input file. It allows you to assign meaningful names to contiguous memory areas on the target board. Memory within such an area shares common attributes:

- Segment Qualifier
- Segment Alignment
- Segment Fill Pattern

Two types of segments can be defined:

- Physical Segments
- Virtual Segment

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.

Physical Segments Example:
For a 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;
```
Linking Issues
Object Allocation

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

Virtual Segment Example:

For an HC12 small memory model, you can define a segment for the direct page area, another one 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;

HC(S)08/RS08 and HC(S)12 Build Tools Utilities Manual
Linking Issues
Object Allocation

```
ROM_AREA = READ_ONLY 0x08000 TO 0xFFFF;
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 5.1 describes the available qualifiers:

Table 5.1 Qualifiers and Their Description

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ_ONLY</td>
<td>Qualifies a segment, where read only access is allowed. Objects within such a segment are initialized at application loading time.</td>
</tr>
<tr>
<td>READ_WRITE</td>
<td>Qualifies a segment, where read and write accesses are allowed. Objects within such a segment are initialized at application startup.</td>
</tr>
<tr>
<td>NO_INIT</td>
<td>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’).</td>
</tr>
<tr>
<td>PAGED</td>
<td>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’).</td>
</tr>
</tbody>
</table>
NOTE For debugging purposes you may want to load code into RAM areas. Because this code should be loaded at load time, such areas should be qualified as READ_ONLY. For the linker, READ_ONLY means that such objects are initialized at program load time. The linker does not know (and does not care) if at runtime the target code writes to a READ_ONLY area.

NOTE Anything located in a READ_WRITE segment is initialized at application startup time. So 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 an 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. You can choose to define your 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:

Table 5.2 Segment Alignment Format

<table>
<thead>
<tr>
<th>Format Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>defaultAlignment</td>
<td>This is the alignment value for all objects which do not match the conditions of any range defined afterward.</td>
</tr>
<tr>
<td>ObjSizeRange</td>
<td>Defines a certain condition. The condition is from the form: size: The rule applies to objects, which size is equal to ‘size’.&lt; size: The rule applies to objects, whose size is smaller than ‘size’.&gt; size: The rule applies to objects, which size is bigger than ‘size’&lt; size: The rule applies to objects, which size is smaller or equal to ‘size’&gt;= 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’.</td>
</tr>
<tr>
<td>alignment</td>
<td>Defines the alignment value for objects matching the condition defined in the current alignment block (enclosed in square bracket).</td>
</tr>
</tbody>
</table>

Segment Alignment 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
```
Linking Issues
Object Allocation

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 the example above:
- In segment DIRECT_RAM, objects whose size is 1 byte are aligned on byte boundary, all other objects are aligned on 2-bytes boundary.
- In segment RAM_AREA, objects whose size is 1 byte are aligned on byte boundary, objects whose 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. You can choose to define your 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 to fill up a segment to the segment end boundary.

Segment Fill Pattern 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 the example above:

- 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 (Freescale (Hiware) + ELF)

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

- Segment Qualifier

Two types of segments can be defined:

- Physical Segments
- Virtual Segment

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.

Physical Segments Example:

For a 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 = READONLY 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.
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.

**Virtual Segment Example:**

For an HC12 small memory model, you can define a segment for the direct page area, another for the rest of the RAM area and another 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 5.1 describes the available qualifiers.

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ_ONLY</td>
<td>Qualifies a segment, where read only access is allowed. Objects within such a segment are initialized at application loading time.</td>
</tr>
<tr>
<td>CODE (ELF)</td>
<td>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 Freescale (former Hiware) object file format use READ_ONLY instead.</td>
</tr>
<tr>
<td>READ_WRITE</td>
<td>Qualifies a segment, where read and write accesses are allowed. Objects within such a segment are initialized at application startup.</td>
</tr>
<tr>
<td>NO_INIT</td>
<td>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').</td>
</tr>
<tr>
<td>PAGED</td>
<td>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').</td>
</tr>
</tbody>
</table>

NOTE For debugging purposes, you may want 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.
Linking Issues
Object Allocation

NOTE Anything located in a READ_WRITE segment is initialized at application startup time. So 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 an 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.

Sequence Enumeration 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;
    DEFAULT_ROM, myCode INTO ROM_AREA;
    SSTACK INTO STK_AREA;
```
In this 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, and so on until all objects are allocated.

**Sequence Enumeration - Further Example:**

```plaintext
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 this 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.

**Figure 5.1 User Defined Sections (.stack)**

<table>
<thead>
<tr>
<th>.data</th>
<th>.bss</th>
<th>User Data 1</th>
<th>...</th>
<th>User Data n</th>
<th>.stack</th>
</tr>
</thead>
</table>
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).

Figure 5.2 User Defined Sections (.txt)

<table>
<thead>
<tr>
<th>.init</th>
<th>.startData</th>
<th>.text</th>
<th>.rodata</th>
<th>.rodata1</th>
<th>User Code1</th>
<th>...</th>
<th>User Code n</th>
<th>.copy</th>
</tr>
</thead>
</table>

**Allocating User Defined Sections (Freescale - 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: Allocate Non-specified Constant Segments in RAM (ELF)`.
- 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:
Linking Issues
Object Allocation

- 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.

Figure 5.3 User Defined Sections (DEFAULT_RAM)

<table>
<thead>
<tr>
<th>DEFAULT_RAM</th>
<th>User Data 1</th>
<th>...</th>
<th>User Data n</th>
<th>SSTACK</th>
</tr>
</thead>
</table>

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).

Figure 5.4 User Defined Sections (DEFAULT_ROM)

<table>
<thead>
<tr>
<th>_PRESTART</th>
<th>STARTUP</th>
<th>ROM_VAR</th>
<th>SSTRING</th>
<th>DEFAULT_ROM</th>
<th>User Code 1</th>
<th>...</th>
<th>User Code n</th>
<th>COPY</th>
</tr>
</thead>
</table>
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 5.4 describes the VECTOR command syntax.

Table 5.4 VECTOR Command Syntax and Descriptions

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VECTOR ADDRESS 0xFFFE 0x1000</td>
<td>indicates that the value 0x1000 must be stored at address 0xFFFE</td>
</tr>
<tr>
<td>VECTOR ADDRESS 0xFFFE FName</td>
<td>indicates that the address of the function FName must be stored at address 0xFFFE.</td>
</tr>
<tr>
<td>VECTOR ADDRESS 0xFFFE FName OFFSET 2</td>
<td>indicates that the address of the function FName incremented by 2 must be stored at address 0xFFFE</td>
</tr>
</tbody>
</table>

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.

You can specify additional entry points using the command `ENTRIES: List of Objects to Link with Application` in the prm file.

**Mandatory Linking of an Object**

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

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

**Mandatory Linking of an Object Example:**

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

In this 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 of all Objects Defined in Object File**

You can choose to link all objects defined in a specified object file in your application.

**Mandatory Linking from All Objects Example:**

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

In this 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

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

Switching Off SmartLinking Example:

```
ENTRIES
*
END
```

In this 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 (Freescale + 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.

You can specify additional entry points using the command `ENTRIES: List of Objects to Link with Application` in the prm file.

Mandatory Linking from an Object

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

This may also be useful for ensuring that a vector table, which has been defined as a constant table of function pointers is linked with the application.
**Mandatory Linking from an Object Example:**

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

In the example above:
- 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**

You can choose to link all objects defined in a specified object file in your application. For that purpose, you need only to specify a `+` after the name of the module in the NAMES block.

**Mandatory Linking from All Objects Example:**

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

In this 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: Additional Object/Library File`. If all binary files should be specified by the command line option `-add`, then an empty NAMES block (just NAMES END) must be specified.

**Names Block Example:**

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

In this 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.

**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 0x007FF;
  ROM_AREA    = READ_ONLY  0x08000 TO 0x0FFFF;
END

PLACEMENT
  myRegister    INTO DIRECT_RAM;
  DEFAULT_RAM   INTO RAM_AREA;
  DEFAULT_ROM   INTO ROM_AREA;
```
In this 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 (Freescale former 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 Additional Object/ Library File. If all binary files should be specified by the command line option -add, then an empty NAMES block (just NAMES END) must be specified.

**Names Block Example:**

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

In this example, the binary files myFile1.o and myFile2.o build the application.
Allocating Variables in “OVERLAYS”

When your application consists of two distinct parts (or execution units) 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.

.prn File Example:

In your source code support you have two execution units: 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 handle 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.

**Overlapping Locals Example:**

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

In this example, the functions f and g are never active at the same time. Therefore the local variables `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. 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 safe side, usually interrupt functions use 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
The 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.

Algorithm Example

```c
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 an interrupt.

```c
... 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.
Linking Issues

Overlapping Locals

Table 5.5  Algorithm Object File Format

<table>
<thead>
<tr>
<th>0x60</th>
<th>0x61</th>
<th>0x62</th>
<th>0x63</th>
<th>0x64</th>
<th>0x65</th>
<th>0x66</th>
<th>0x67</th>
<th>0x68</th>
</tr>
</thead>
<tbody>
<tr>
<td>g_par</td>
<td>ch</td>
<td>inter_loc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l_par</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</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-SmartLinked 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, it now also needs locals which must be alive over the call to main, define the _Startup function as a 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 include parameters not passed in registers as well), the matching is done by the linker using the parameter name.

Consider the following example:

```c
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 callee's argument is done by the name, they both have to use the same name. Because of this, the compiler generates an artificial name for the callee’s parameter `_callee_0`. 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 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 built 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.
Linking Issues

Overlapping Locals

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):

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

This generates 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 a Function with Overlapping Parameters in Assembler

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

For example, we want to define the function callee:

```asm
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:

```asm
_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
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:

  ```assembly
  _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 of two functions put into one segment:

  ```assembly
  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:
Linking Issues

Overlapping Locals

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

Example of an invalid non exported definition of a parameter:

loc1: DS 4
code_seg0: SECTION
callee0: CLEAR loc0,4 RETURN
code_seg1: SECTION
callee1: CLEAR loc1,4 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
_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
Linking Issues

Overlapping Locals

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 invalid usage of a local variable:

```assembly
_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:

```assembly
_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 Freescale (former 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 so this restriction does not exist.

The following example does not work in the Freescale 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 this, 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 Section in Map File**

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

**DEPENDENCY TREE Example:**

```c
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
```
Linking Issues

Overlapping Locals

| * 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:

Recursion Checks Example

```c
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
```
Linking Issues  
Overlapping Locals

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:

```
DEPENDENCY 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.
Linking Issues

**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 and 0 is taken as address of this object.

**NOTE** There is no warning issued for predefined segments like SSTACK or OVERLAP, even if these segments are empty and not explicitly allocated. The warning is only issued for user defined segments.

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

Example:

With the following C source code:

```c
#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:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>__SEG_START_SSTACK</td>
<td>0x400</td>
</tr>
<tr>
<td>__SEG_END_SSTACK</td>
<td>0x500</td>
</tr>
<tr>
<td>__SEG_SIZE_SSTACK</td>
<td>0x100</td>
</tr>
</tbody>
</table>

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:

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

Link parameter File:

```text
SECTIONS
...
  MY_ROM0  = READ_ONLY 0x06000 TO 0x07FFF;
  MY_ROM1  = READ_ONLY 0x18000 TO 0x18FFF;
```
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:

```c
#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(); ... }
```

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 three functions, but thousands or 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.
Linking Issues
Automatic Distribution of Paged Functions

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: Enable Distribution Optimization (ELF). 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: Specify Distribution File Name (ELF). 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_SEGS 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 on have to be rerun (or you must be sure not 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 below: 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:

```c
#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 those 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: Generate Distribution Information File (ELF) 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: Generate Map File 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”.

Requirements:

The explained method only works with a 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 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>
<thead>
<tr>
<th>Method</th>
<th>prm File Controlled</th>
<th>Automatic Linker Controlled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>Needs some configuration prm file needs adaptations</td>
<td>Easy to use</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Just call _Checksum_Check</td>
</tr>
<tr>
<td>Robustness</td>
<td>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.</td>
<td>Good. Nothing (or few things) to configure</td>
</tr>
<tr>
<td>Control</td>
<td>Everything is in full user control.</td>
<td>Poor. Only if a segment should be checked can be controlled.</td>
</tr>
<tr>
<td>Target Memory Usage</td>
<td>Good, only what is needed is present.</td>
<td>Needs more memory because of the control data structure.</td>
</tr>
<tr>
<td>Execution Time.</td>
<td>Mainly depends on method. Too much might be checked as the code size is not exactly known.</td>
<td>Mainly depends on method. Only needed areas are checked.</td>
</tr>
</tbody>
</table>
Linking Issues
Checksum Computation

**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):

```plaintext
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: Checksum Computation (ELF)] 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 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 provides this information similar to the way it provides copy down and zero out information.

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

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

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

```c
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 Structure Detection

The linker reads 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 always matches the structure generated by the linker.

The linker knows 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, it is 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
An Assembly application can be linked using the prm file or SmartLinking, while warnings can be ignored.

prm File
When an application consists of 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.

prm File Example:

```
LINK   test.abs
NAMES  test.o test2.o END
SECTIONS
   DIRECT_RAM = READ_WRITE 0x00000 TO 0x0FF;
```
Linking Issues

Linking an Assembly Application

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 0xFFFFE Start ; Initialize Reset Vector

In this 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:

‘s 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.

SmartLinking Example:

Assembly source file

XDEF entry
dataSec1: SECTION
data1:     DS.W 1
dataSec2: SECTION
data2:     DS.W 2
Linking Issues

Linking an Assembly Application

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 this example:
- The function entry is defined 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 is never referenced.

You can choose to switch smart linking OFF for your 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:

LINK test.abs
NAMES test.o END
In this example:

- The function entry is defined 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.
LINKING ISSUES

Linking an Assembly Application

LINK_INFO (ELF)

Some compilers support writing additional information into the ELF file. This information consists 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 different content. If so, the linker issues a warning.

Finally, the linker issues all LINK_INFOs into the generated output ELF file.

This feature can be used to warn you 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.
SmartLinker Parameter File

The SmartLinker’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 for the examples included in your installation.

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
where:
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 | STATISTIC
Dependency= ROOT {ObjName} END
| ObjName USES {ObjName} END
| ObjName ADDUSE {ObjName} END
```

HC(S)08/RS08 and HC(S)12 Build Tools Utilities Manual 197
SmartLinker Parameter File
Syntax of the Parameter File

| ObjName DELUSE {ObjName} END |
| SegmentDef= SegmentName "=" SegmentSpec ";" |
| SegmentName= Ident |
| SegmentSpec= StorageDevice Relocation 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 [SIZE <Size>] | METHOD_XOR |
| MemoryArea= StorageDevice Range |
| StorageDevice= READ_ONLY | CODE | READ_WRITE | PAGED | NO_INIT |
| Range= exp (TO | SIZE) exp |
| Relocation= RELOCATE_TO Address |
| 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) |
| 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 can appear anywhere in a parameter file, except where file names are expected.
You can use either C style comments or Modula-2 style comments.
To keep your sources portable, file names should not contain paths. 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. However you should make sure that the SEGMENTS block is specified before the PLACEMENT block.
There are some sections named .data, .text, .stack, .copy, .rodata, .startData, and .init. Information about these sections can be found in the chapter on 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 reserves the address range from 0x800 to 0x8FF for allocation of variables.
```
DEFAULT_RAM INTO READ_WRITE 0x800 TO 0x8FF;
```
The INCLUDE Directive

A special directive INCLUDE allows you 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.

**Include Directive Example:**

```plaintext
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) */
  minil.o
```
SmartLinker Commands

This section describes each SmartLinker parameter command. Each command description includes the following:

- **Syntax**: Description of the command syntax.
- **Description**: Detailed description of the command.
- **Example**: Example of how to use the command.

Some commands are available only in ELF/DWARF format, and some commands only in Freescale (former Hiware) object file format. This is indicated with the object file format in parenthesis (ELF) or (Freescale).

If a command is available only for a specific language, it is also indicated. For example, ‘M2’ denotes that the feature is available only for Modula-2 linker parameter files. Additionally, it is also noted if the behavior of a command is different for Freescale and ELF/DWARF formats.

**AUTO_LOAD: Load Imported Modules (Freescale, 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-2 applications. 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 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 [SIZE <Size>] | 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  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.

For example:

SEGMENTS
MY_ROM = READ_ONLY 0xE020 TO 0xFEFF FILL 0xFF;
The checksum can only be 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 optional SIZE argument. If the SIZE option is not specified, the size of the INTO_AREA is used instead.
- 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 value 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 must 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

Assume the following memory content:
0x1000 02 02 03 04
Then the XOR 1 byte checksum from 0x1000 to 0x1003 is 0x07
(=0x02^0x02^0x03^0x04).

NOTE
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.

NOTE
METHOD_XOR/METHOD_ADD also support computing the checksum with larger element sizes.

By default, the element size is taken as the size of the INTO MemoryArea part.
However for the METHOD_ADD the size can also be explicitly specified (in bytes) to be less than the INTO MemoryArea part’s size.
With a element size of 2, the checksum of the example would be 0x0506 (= 0x0202 ^ 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: Check Module Keys (Freescale, 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: Specify the RAM Start (Freescale)

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
SmartLinker Commands

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 anymore.

Example
START 0x1000

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 Keyword

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: List of Objects to Link with Application section. A ROOT entry with several objects is semantically the same as an OVERLAP_GROUP: Application Uses.
Overlapping (ELF) 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:

```c
void main(void) { int i; ....} 
void interrupt int1(void) { int j; ...} 
void interrupt int2(void) { int k; ...} 
```

prm file:

```plaintext
...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 Keyword

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:

```c
void f(void(*)(void))(int i; ...fct();...)
void g(void) { int j;...}
void h(void) { int k;...}
void main(void) { f(g); f(h); }
```

prm file:

```plaintext
DEPENDENCY
    f USES g h END
```
SmartLinker Commands

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 Keyword

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:

```c
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:

```prm
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:

```c
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 Keyword**

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 DEPENDENCY 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:

```c
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 overlapping.
See Also:
Keyword OVERLAP_GROUP: Application Uses Overlapping (Elf)

ENTRIES: List of Objects to Link with Application

Syntax (Elf):

ENTRIES

[FileName " :"](*|objName)

{{FileName ";"}](*|objName)}

END

Syntax (Freescale - Hiware):

ENTRIES objName {objName} END

Description

The ENTRIES block is optional in a prm file and it cannot be specified more than once. 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 7.1 describes the notations that are supported.

Table 7.1 Notations and Their Description

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Object Name&gt;</td>
<td>The specified global object must be linked with the application</td>
</tr>
<tr>
<td>&lt;FileName&gt;:&lt;Object Name&gt; (Elf)</td>
<td>The specified local object defined in the specified binary file must be linked with the application</td>
</tr>
<tr>
<td>&lt;FileName&gt;:* (Elf)</td>
<td>All objects defined within the specified file must be linked with the application</td>
</tr>
<tr>
<td>* (Elf)</td>
<td>All objects must be linked with the application. This switches OFF smart linking for the application</td>
</tr>
</tbody>
</table>
ELF Specific Issues:

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 this 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 this 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: Application Has Banked Data (Freescale)

Syntax

HAS_BANKED_DATA
SmartLinker Commands

**Description**
This entry is used to specify for the HC12 in the Freescale 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: Link Hex File with 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 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 Freescale 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 Freescale (former 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: Specify Application Init Point**

**Syntax**

```plaintext
INIT FuncName
```

**Description**

The INIT command is mandatory for assembly application and optional otherwise. It cannot be specified more than once 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:**

You can specify any static or global function as entry point.

**Example**

```plaintext
INIT MyGlobStart /* Specify a global variable as application entry point.*/
```

**ELF Specific Example:**

```plaintext
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: Specify Name of Output File**

**Syntax**

```plaintext
LINK <NameOfABSFile> ['AS ROM_LIB']
```
**SmartLinker Commands**

**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**: Absolute Path 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**: Text Path 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**: Define Absolute File Name 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, it has higher priority than the LINK command.

**Example**

```
LINK fibo.abs

NAMES  fibo.o startup.o END

SECTIONS
  MY_RAM  =  READ_WRITE  0x1000 TO 0x18FF;
  MY_ROM  =  READONLY  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: Name of Application Root Function

Syntax

MAIN FuncName

Description

The MAIN command is optional and cannot be specified more than once 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:

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:

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 Libraries prm file, a warning is generated.

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)

HC(S)08/RS08 and HC(S)12 Build Tools Utilities Manual 215
SmartLinker Commands

LLOC

|OBJ_DEP|OBJ_UNUSED|COPYDOWN|OVERLAP_TREE|STATISTIC|MODULER_STATISTIC|

Syntax (Freescale):

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 7.2 describes the map file specifiers that are available.

Table 7.2  Map File Specifiers and Their Description

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL (ELF)</td>
<td>A map file must be generated containing all information available</td>
</tr>
<tr>
<td>COPYDOWN (ELF)</td>
<td>The information about the initialization value for objects allocated in RAM must be written to the map file (Section COPYDOWN in the map file)</td>
</tr>
<tr>
<td>FILE (ELF)</td>
<td>The information about the files building the application must be inserted in the map file (Section FILE in the map file).</td>
</tr>
<tr>
<td>NONE (ELF)</td>
<td>No map file must be generated</td>
</tr>
<tr>
<td>OBJ_ALLOC (ELF)</td>
<td>The information about the allocated objects must be inserted in the map file (Section OBJECT ALLOCATION in the map file)</td>
</tr>
<tr>
<td>SORTED_OBJECT_LIST (ELF)</td>
<td>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)</td>
</tr>
<tr>
<td>OBJ_UNUSED (ELF)</td>
<td>The list of all unused objects must be inserted in the map file (Section UNUSED OBJECTS in the map file)</td>
</tr>
<tr>
<td>OBJ_DEP (ELF)</td>
<td>The dependencies between the objects in the application must be inserted in the map file (Section OBJECT DEPENDENCY in the map file).</td>
</tr>
</tbody>
</table>
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
```
SmartLinker Commands

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 Freescale (former Hiware) format prm file, following commands are also supported:
MAPFILE OFF is equivalent to MAPFILE NONE
MAPFILE ON is equivalent to MAPFILE ALL

NAMES: List 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: Object File Path and finally in the directories specified in the environment variable GENPATH: Define Paths to Search for Input Files. The files may be either object files, absolute or ROM Library files or libraries.

Additional files may be specified by the option -Add: Additional Object/Library File. 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 this example, the application fibo is build from the files ‘fibo.o’ and ‘startup.o’.

**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 allows you to specify a group of functions which do not overlap.
OVERLAP_GROUP is only available in the ELF object file format. However, the same functionality can be achieved with the DEPENDENCY: Dependency Control, ROOT Keyword command, which is also available in the Freescale 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 built 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_GROUP: Application Uses Overlapping (ELF) 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: Dependency Control

PLACEMENT: Place Sections into Segments

Syntax (ELF)

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 more than once.

Each placement statement between the PLACEMENT and END defines:

- (ELF) a relation between logical sections and physical memory ranges called segments.
- (Freescale) a relation between logical segments and physical memory ranges called sections. Standard terminology for Freescale (former Hiware) uses a SECTIONS block, rather than a SEGMENTS block; this syntax should be accepted by the ELF linker.

Example (ELF)

SEGMENTS
ROM_1 = READ_ONLY 0x800 TO 0xAF;
SmartLinker Commands

ROM_2  = READ_ONLY 0xB00 TO 0xCFF;
END
PLACEMENT
  DEFAULT_ROM INTO ROM_1, ROM_2;
END

In this 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 SEGMENTS block must always be defined before the PLACEMENT block, because the segments referenced in the PLACEMENT block must previously be defined in the SEGMENTS 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 .prn 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.

Example (Freescale)

SECTIONS
  ROM_1  = READ_ONLY 0x800 TO 0xAFF;
PLACEMENT
  DEFAULT_ROM, ROM_VAR INTO ROM_1;
END

In this example, the objects from segment ‘DEFAULT_ROM’ are allocated first and then the objects from segment ‘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 section is full (i.e., the next object to be allocated doesn’t fit anymore), allocation continues in the next section in the list.
PRESTART: Application Prestart Code (Freescale)

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: Define Memory Map (Freescale)

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 more than once. The SECTIONS block must be directly followed by the PLACEMENT block.
The SECTIONS command allows you 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 (Qualifier Handling).
- a start and end address or a start address and a size.

**NOTE** The ELF linker accepts SECTION syntax as an alias for SEGMENTS syntax.

### Section Qualifier

The following qualifiers are available for sections:

- **READ_ONLY**: used for address ranges which are initialized at program load time. The application (*.abs) contains content only 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 on 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 speed up 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 your responsibility to select the correct page before accessing the data allocated on a certain page.
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 constants in READ_WRITE sections also need RAM memory, and 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 details 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.

### Table 7.3 Section Qualifiers

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Initialized Variables</th>
<th>Non-Initialized Variables</th>
<th>Constants</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ_ONLY</td>
<td>Not applicable (1)</td>
<td>Not applicable (1)</td>
<td>Content written to target address</td>
<td>Content written to target address</td>
</tr>
<tr>
<td>READ_WRITE</td>
<td>Content written into copy down area, together with info where to copy it at startup. Area contained in zero out information (3) (4)</td>
<td>Area contained in zero out information (4)</td>
<td>Content written into copy down area, together with info where to copy it at startup. Area contained in zero out information (3) (4)</td>
<td>Not applicable (1) (2)</td>
</tr>
<tr>
<td>NO_INIT</td>
<td>Not applicable (1)</td>
<td>Just handled as allocated. Nothing generated.</td>
<td>Not applicable (1)</td>
<td>Not applicable (1)</td>
</tr>
<tr>
<td>PAGED</td>
<td>Not applicable (1)</td>
<td>Just handled as allocated. Nothing generated.</td>
<td>Not applicable (1)</td>
<td>Not applicable (1)</td>
</tr>
</tbody>
</table>
**Example**

```plaintext
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 this example:

- The section ‘ROM’ is a READ_ONLY memory area. It starts at address 0x1000 and its size is 0x2000 bytes (from address 0x1000 to 0x2FFF).
- The section ‘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 section ‘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 sections ‘Page0’ and ‘Page1’ are READ_WRITE memory areas. These are overlapping segments. It is your 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: Define Memory Map (ELF)

Syntax

```
SEGMENTS {(READ_ONLY|READ_WRITE|NO_INIT|PAGED)
           <startAddr> (TO <endAddr> | SIZE <size>)
           [RELOCATE_TO Address]
           [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 more than once. The SEGMENTS command allows you 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 relocation rule
- an optional alignment rule
- an optional fill pattern.
- optional constant optimization with Common Code commands.

Segment Qualifier

The following qualifiers 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.
SmartLinker Commands

- **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 speed up application startup.

- **PAGED**: used for address range 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 your responsibility to select the correct page before accessing the data allocated on a certain page.

## Qualifier Handling

Table 7.4 Qualifier Handling

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Initialized Variables</th>
<th>Non-Initialized Variables</th>
<th>Constants</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ_ONLY</td>
<td>Not applicable (1)</td>
<td>Not applicable (1)</td>
<td>Content written to target address</td>
<td>Content written to target address</td>
</tr>
<tr>
<td>READ_WRITE</td>
<td>Content written into copy down area, together with info where to copy it at startup. Area contained in zero out information (3) (4)</td>
<td>Area contained in zero out information (4)</td>
<td>Content written into copy down area, together with info where to copy it at startup. Area contained in zero out information (3) (4)</td>
<td>Not applicable (1) (2)</td>
</tr>
<tr>
<td>NO_INIT</td>
<td>Not applicable (1)</td>
<td>Just handled as allocated. Nothing generated.</td>
<td>Not applicable (1)</td>
<td>Not applicable (1)</td>
</tr>
<tr>
<td>PAGED</td>
<td>Not applicable (1)</td>
<td>Just handled as allocated. Nothing generated.</td>
<td>Not applicable (1)</td>
<td>Not applicable (1)</td>
</tr>
</tbody>
</table>
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 constants in READ_WRITE sections also need RAM memory, and 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 details 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 this 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’ are READ_WRITE memory areas. These are overlapping segments. It is your responsibility to select the correct page before accessing any data allocated in one of these segments. Variables allocated in this segment will not be initialized at application startup.
Defining a Relocation Rule

The relocation rule can be used if a segment is moved to a different location at runtime. With the relocation rule, the linker can be instructed to use a different runtime addresses for all objects in a segment.

This is useful when at runtime the code is copied and executed at a different address than its is linked to. One example is a flash programmer which has to run out of RAM. Another example is a boot loader which does move the actual application to a different address before running it.

An relocation rule can be specified as follows:

```
RELOCATE_TO Address
```

**Address:** is used to specify the runtime address of the object.

Example

```
SEGMENTS
    CODE_RELOC  = READ_ONLY 0x8000 TO 0x8FFF RELOCATE_TO 0x1000;
...
END
```

In this example, references to functions in `CODE_RELOC` will use addresses in the 0x1000 up to 0x1FFF area. But the code will be programmed at 0x8000 up to 0x8FFF.

With `RELOCATE_TO` code can be executed at a different address than it was allocated. The code does not need to be position independent (PIC). However if the code is not PIC, it usually will not run at its allocation address as all references inside of this code do also refer to the `RELOCATE_TO` address.

**NOTE** Usually the `RELOCATE_TO` address is in RAM. The linker does not check for overlaps in the `RELOCATE_TO` address area. Instead the prm file should be setup that no such overlapping is possible.
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:

\[
\text{ALIGN \{defaultAlignment\} \{[\{'\{<Number> | TO \<Number>\} | \{'\<Number> | \'>\ | \'<\ | \'=\} TO \<Number>\} | :<alignment>\}}
\]

*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 this 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

A fill pattern can be associated with each segment in the application. This can be useful for automatically initializing uninitialized variables in the segments with a predefined pattern.

A fill pattern can be specified as follows:

```
FILL <HexByte> {<HexByte>}
```

**NOTE** Any segment defined in the SEGMENTS portion of the prm file with the FILL command will be filled only if the segment is also used in the PLACEMENT section of the prm file. If necessary, add a dummy entry to the PLACEMENT section.

**Example**

```
SEGMENTS
   RAM_1  = READ_WRITE 0x800 TO 0x8FF
         FILL 0xAA 0x55;
END
PLACEMENT
   DUMMY INTO RAM_1
END
```

In this example, uninitialized 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 times as required to fill the objects. In this 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 this example, an object whose 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 this example, uninitialized 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 for assigning 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

```c
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 Freescale (former Hiware) format, the linker only optimizes 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`.

### Examples

**C-Source File:**

```c
void print1(void) {
    printf("Hello");
}
void print2(void) {
    printf("Hello");
}
```

**Prm File:**

```c
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 is 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 not, however, 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 the compiler optimizations can produce smaller applications, if the compiler optimizations 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: Define Stack Size

**Syntax**

```
STACKSIZE Number
```

**Description**

The STACKSIZE command is optional in a prm file and it cannot be specified more than once. Additionally, you cannot specify both STACKTOP: Define Stack Pointer Initial Value 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 through a STACKSIZE command alone, the stack is placed next to the section `.data`

**NOTE**

In the Freescale (former 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 this 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 through 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 this example, the section SSTACK is allocated from address 0xB5F down to address 0xB00. The stack initial value is set to 0xB5E.
STACKTOP: Define Stack Pointer Initial Value

Syntax

STACKTOP Number

Description

The STACKTOP command is optional in a prm file and it cannot be specified more than once. Additionally, you cannot specify both STACKTOP and STACKSIZE: Define Stack Size 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 through 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 through 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 0xAF0;
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 this example, the stack pointer will be defined from address 0xB7E down to address 0xB00.
START: Specify the ROM Start (Freescale)

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 anymore.

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: Initialize Vector Table

Syntax
VECTOR (InitByAddr | InitByNumber)

Description
The VECTOR command is optional in a prm file and it can be specified more than once.

A vector is a small piece of memory, having the size of a function address. This command allows you 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).
SmartLinker Commands

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 \(<Number> \times <Size of a Function Pointer>\).
  - For targets with vectors located from 0xFFFE and allocated downwards, VECTOR 0 maps to 0xFFFE. In general the address is 0xFFFE - \(<Number> \times 2\).
  - 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 this 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 this 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.
ELF Sections

Using the section concept gives you 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 sections 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.

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.

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 (see the Compiler or Assembler manual). The second is done in the parameter file using the SEGMENTS and PLACEMENT commands.

Some Predefined Sections are handled in a particular way.
Predefined Sections

There are several predefined section names which can be grouped into sections named by the runtime routines:

- Sections for things other 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.

The following paragraphs describe each of these predefined sections.

.rodata1

This predefined section is all string literals. For example, ("This is a string") is 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.
ELF Sections

Section

**.stack**

The runtime stack has its own segment named `.stack`. It should always be allocated to a READ_WRITE segment.

**.data**

This predefined section 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:

![Memory Inclusion Order for .data](image)

**.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`. If any of the sections `.rodata`, `.rodata1`, `.startData`, or `.init` is not associated with a segment, these sections are included in the `.text` memory area.

**.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. This section must be associated with a NO_INIT segment.

NOTE The .data and .text sections must always be associated with a segment.
Segments

Using the segment concept gives you 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.

Segments and Sections

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.

Segment

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 (see the Compiler or Assembler manual). The second is done in the parameter file using the SECTIONS and PLACEMENT commands (see: The Syntax of the Parameter File).

Some predefined segments are handled in a particular way.
Predefined Segments

There are several predefined segment names which can be grouped into segments named by the runtime routines:

- Segments for things other 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.

The following paragraphs describe 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:

```c
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 DEFAULT_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.
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.

**NOTE** The DEFAULT_RAM and DEFAULT_ROM segments must always be associated with a section.
Examples of Using Sections

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:

```plaintext
LINK first.ABS
NAMES first.o strings.o startup.o END
STACKSIZE 0x200
SECTIONS
   ROM1 = READ_ONLY 0x4000 TO 0xFFF;
   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:

```plaintext
/* 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:

```plaintext
LINK bufram.ABS
NAMES
   bufram.o startup.o
END
STACKSIZE 0x200
```
Examples of Using Sections

Example 2

```c
SECTIONS
    BatteryRAM = NO_INIT 0x1000 TO 0x13FF;
    MyRAM = READ_WRITE 0x5000 TO 0x5FFF;
PLACEMENT
    DEFAULT_ROM INTO READONLY 0x2000 TO 0x2800;
    DEFAULT_RAM INTO MyRAM;
    Buffered_RAM INTO BatteryRAM;
END
```
Program Startup

This section deals with advanced material. First time users may skip this section; standard startup modules taking care of the common cases are delivered with the 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 readme.txt in the LIB directory subfolders.

NOTE The code shown in this chapter is example code. To understand what the startup modules for your environment do, be sure to look at the files in the installation.

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.

There are standard startup routines for every processor and memory model. They are 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 differ only in the way they make use of that information.

Startup Descriptor (ELF)

The startup descriptor of the linker is declared in code similar to that shown below. Note that depending on architecture or memory model your startup descriptor may be different.

```c
typedef struct {
    unsigned char *_FAR beg; int size;
} _Range;

typedef struct _Copy {
    int size; unsigned char * far dest;
} _Copy;

typedef void (*_PFunc)(void);
```
typedef struct_LibInit {  
  _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 contain some flags which can be used to detect special conditions at startup. Currently two bits are used, as shown in Table 11.1:

Table 11.1 Bit Description

<table>
<thead>
<tr>
<th>Bit #</th>
<th>Set if...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The application has been linked as a ROM library</td>
</tr>
<tr>
<td>1</td>
<td>There is no stack specification</td>
</tr>
</tbody>
</table>

Table 11.1 Bit Description
Bit 1 (with mask 2) is tested in the startup code to detect if the stack pointer should be initialized.

**main**

This 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**

This is valid only if `(flags & 2) == 0`. This field contains the initial value of the stack pointer.

**nofZeroOuts**

This 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**

This field 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:

```plaintext
CopyData = {Size\[t\] TargetAddr (Byte)\[t\] Size Alignment} \(0x0\[t\].
```

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 processor’s capability to access non aligned data. For small processors, there is usually no alignment. The size \(t\) of `Size\[t\]` and `0x0\[t\]` depends on the target processor and memory model.
Program Startup
Startup Descriptor (ELF)

nofLibInits
This 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
This 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
This 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
This field 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
This 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
This 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)

You can define your own startup structure. That means you can remove the fields, which are not required for your application, or move the fields inside of the structure. If you change the startup structure, it is your responsibility to adapt the startup function to match the modification.

Example

If you do not have any RAM area to initialize at startup, no ROM libraries and no C++ modules in the application, you can define the startup structure as follows:

```c
extern struct _tagStartup {
    unsigned short  flags;
    _PFunc          main;
    unsigned short  stackOffset;
} _startupData;
```

In that case the startup code must be adapted in the following way:

```c
extern void near _Startup(void) {
    /* purpose: 1) initialize the stack
     2) call main;
     parameters: NONE */
    do { /* forever: initialize the program; call the root-procedure */
        INIT_SP_FROM_STARTUP_DESC();
        /* 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. You are free to remove fields inside of the structure, but you should respect the names of the different fields; if you do not, 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 can 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 a name other than `_Startup` and define it as the entry point for your application using the command `INIT`:

  ```
  INIT function_name
  ``

  where `function_name` is the startup routine.

Startup Descriptor (Freescale)

The Freescale (former Hiware) startup descriptor of the linker is declared as below. Note that depending on architecture or memory model it may be a little bit different.

```c
typedef struct{
    unsigned char  *beg; int size;
} _Range;

t 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:

```c
struct _tagStartup _startupData;
```

The linker expects somewhere in your application a declaration of the variable `_startupData`, that is:

```c
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

This field contains some flags, which may be used to detect special conditions at startup. Currently two bits are used, as shown in Table 11.2:

<table>
<thead>
<tr>
<th>Bit #</th>
<th>Set if...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The application has been linked as a ROM library</td>
</tr>
<tr>
<td>1</td>
<td>There is no stack specification</td>
</tr>
</tbody>
</table>

This flag is tested in the startup code to detect if the stack pointer should be initialized.

main

This 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

This is used only for processors having paged memory and memory models supporting only one page. In this case, dataPage gives the page.

stackOffset

This is valid only if flags == 0. This field contains the initial value of the stack pointer.

nofZeroOuts

This is the number of READ_WRITE segments to fill with zero bytes at startup.

pZeroOut

This 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

This 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:

\[
\text{CopyData} = \{\text{Size}_{[2]} \ TargetAddr \ (\text{Byte})^{\text{Size}} \ 0x0}_{[2]} \]

The size is a binary number whose most significant byte is stored first.

**libInits**
This 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**
This 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 (Freescale)**

There are two ways to replace the standard startup routine with one of your own:

- You can 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 a name other than _Startup and define it as the entry point for your application using the command INIT:

```
INIT function_name
```

where 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 can use this startup for your own modifications or just to get familiar with the startup concept.

The code shown here is an example and may be different depending on the actual implementation. Please see the files in your installation directory.

---

**Listing 11.1  Header File startup.h:**

```c
/*------------------------------------------------------------------------------------------
 FILE : startup.h
 PURPOSE : data structures for startup
 LANGUAGE: ANSI-C
--------------------------------------------------------------------------------------------

#ifndef STARTUP_H
#define STARTUP_H
#ifdef __cplusplus

#define STARTUP_H
#define define STARTUP_H
#define __cplusplus

```

---

HC(S)08/RS08 and HC(S)12 Build Tools Utilities Manual
extern "C" {
#endif
#include <stdtypes.h>
#include <hidef.h>
/*
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 */
#if define __ELF_OBJECT_FILE_FORMAT__
/* ELF/DWARF object file format */
/* attention: the linker scans for these 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 */
};
*/
**Program Startup**

*User-Defined Startup Routines (Freescale)*

```c
_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;    /* ptr 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 */
/*-------------------------------------------------*/
#endif /* STARTUP_H */

Listing 11.2 Implementation File startup.c:

```c
#include <hidef.h>
#include <startup.h>

/******************************************************************************/
FILE     : startup.c
PURPOSE  : standard startup code
LANGUAGE : ANSI-C / HLI
******************************************************************************/
#define __cplusplus
#endif
#endif /* STARTUP_H */

struct _tagStartup _startupData; /* startup info */

/******************************************************************************/
/* purpose: zero out RAM-areas where data is allocated. */
int i, j;
unsigned char *dst,
_Range *r;
unsigned char *pZeroOut;
for (i=0; i<_startupData->nofZeroOuts; i++) {
  dst = r->beg;
  j = r->size;
```
do {
    *dst = ‘\0’; /* zero out */
    dst++;
    j--;
} while(j>0);
}

/*----------------------------------------------------*/
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__
    #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

Program Startup
User-Defined Startup Routines (Freescale)

```c
_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);
    }
} #endif
```
Program Startup
User-Defined Startup Routines (Freescale)

/*-------------------------------------------------------*/
#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 */
} /*-----------------------------------------------*/
Program Startup
User-Defined Startup Routines (Freescale)
The Map File

If linking succeeds, a protocol of the link process is written to a list file called a 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: Text Path.

Map File Contents

The map file consists of the following 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.

**NOTE**

If linking fails because there are objects which were not found in any object file, no map file is written.
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.

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 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.

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 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.

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 (NAMES: List Files Building the Application) 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.

Listing 13.1 Header File Example:

```c
/* 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.
Listing 13.2 Startup Descriptor Example

/* 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.

Listing 13.3 Linker Parameter File Example

/* 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.

**Listing 13.4 Map File Example**

```
*************** 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 **********************
```
Now we want to use the ROM library from our application, as follows:

**Listing 13.5 Simple Application Example**

```c
/* 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:

**Listing 13.6 Linking Example**

```bash
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 may 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):

**Listing 13.7 Map File After Linking Example**

```c
extern struct __Startup{
    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
};
```
### ROM Libraries

#### Using ROM Libraries

<table>
<thead>
<tr>
<th>Variables</th>
<th>Address</th>
<th>Size</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnt</td>
<td>5000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>__startupData</td>
<td>1000</td>
<td>2C</td>
<td>@</td>
</tr>
<tr>
<td>RL_counter</td>
<td>4000</td>
<td>2</td>
<td>@</td>
</tr>
<tr>
<td>_startupData</td>
<td>600A</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

**Module:**

- **X:\FREESCALE\DEMO\M68KC\rl.o**
  - **PROCEDURES:**
    - RL_Count
      - **Address:** 102E
      - **Size:** 14
      - **Field:** @
    - __startupData
      - **Address:** 1000
      - **Size:** 2C
      - **Field:** @
    - RL_counter
      - **Address:** 4000
      - **Size:** 2
      - **Field:** @

**Module:**

- **startup.o**
  - **PROCEDURES:**
    - ZeroOut
      - **Address:** 6062
      - **Size:** 50
    - CopyDown
      - **Address:** 60B2
      - **Size:** 54
    - ProcessStartupDesc
      - **Address:** 6142
      - **Size:** 3E
    - HandleRomLibraries
      - **Address:** 6106
      - **Size:** 3C
    - Start
      - **Address:** 6180
      - **Size:** 20
    - _Startup
      - **Address:** 61A0
      - **Size:** 20
  - **VARIABLES:**
    - __startupData
      - **Address:** 600A
      - **Size:** 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 an EEPROM) separately.
ROM Libraries
Using ROM Libraries
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 initializing 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. You can decide if you want 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
How To Initialize the Vector Table

### Example:

```assembly
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
```

### Notes

- The functions ‘IRQFunc’, ‘XIRQFunc’, ‘SWIFunc’, ‘OpCodeFunc’, ‘ResetFunc’ are published. This is required because they are referenced in the linker prm file.

- 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.

- All Interrupt functions must be terminated with an RTI instruction.

The vector table is initialized using the linker command VECTOR ADDRESS.
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:
How To...  
How To Initialize the Vector Table

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  
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_ROM 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
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 bits 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.
How To Initialize the Vector Table

Example:

```assembly
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 SmartLinker links only the referenced objects in the absolute file.
Burner Utility

Introduction

The CodeWarrior IDE burner utility converts an .ABS file into a file that can be handled by an EPROM burner. The Burner is available as either:

- An interactive burner with a graphical user interface (GUI).
- A batch burner that either accepts commands from a command line or in a command file. It can then be invoked by the Make Utility.

This section consists of the following chapters:

- Interactive Burner (GUI): Description of GUI
- Batch Burner: Description of Batch Burner Language (BBL)
- Burner Options: Options you can use to control the application
- Environment Variables: Environment variables used by the application
- Burner Messages: Messages produced by the Application
Product Highlights

The burner utility has:

- Powerful User Interface
- On-line Help
- Flexible Message Management
- 32bit Application
- Generation of S-Record files, Binary or Intel Hex files
- Splitting up application into different EEPROMS (1, 2 or 4 bytes bus width)
- Both interactive (GUI) and batch language interface (Batch Burner)
- Batch Burner Language with: baudRate, busWidth, CLOSE, dataBit, destination, DO, ECHO, ELSE, END, FOR, format, header, IF, len, OPENCOM, OPENFILE, origin, parity, PAUSE, SENDBYTE, SENDWORD, SLINELEN, SRECORD, swapByte, THEN, TO, undefByte.
- Supports Freescale (former Hiware) and ELF/DWARF Object File Format, S-Records and Intel Hex Files as input
- Supports a serial programmer attached to serial port with various configuration settings
- Powerful batch burner language with various commands (fillByte, origin, destination, range, baudRate, header,...)

Starting the Burner Utility

All of the utilities described in this book may be started from executable files located in the Prog folder of your IDE installation. The executable files are:

- linker.exe The SmartLinker
- maker.exe Maker: The Make Tool
- burner.exe The Burner Utility
- decoder.exe The Decoder
- libmaker.exe Libmaker

With a standard full installation of the HC(S)08/RS08 CodeWarrior IDE, the executable files are located at:

C:\Program Files\Freescale\CW08 v5.x\Prog
To start the Burner Utility, you can click on burner.exe. The Burner Default Configuration window appears:

Figure 14.1 Burner Default Configuration Window
Alternatively, from within the IDE Project Target Settings window, select the Burner for HC08 option from among the listed settings panels. Click on the Burner command button in the Burner for HC08 panel. The Burner window of the Interactive Burner (GUI) appears:

Figure 14.2 IDE Project Target Settings Panel Burner for HC08 Window
Interactive Burner (GUI)

You can use the interactive burner user interface (GUI) to burn your EEPROM instead of writing a batch burner language file. Within the GUI you can set all parameters and receive the output needed for a batch burner language file. You can then use the commands displayed on the Burner Dialog Box Command File tab in a make file.

Burner Default Configuration Window

When you start the Burner, the Burner Default Configuration window opens.

Figure 15.1 Burner Default Configuration Window

To open the burner dialog box, click the Burner Dialog icon in the tool bar or select the Burner Dialog option from the Burner drop-down menu.

The burner dialog box can also be accessed by the following command line option:

`burner.exe -D`
Burner Dialog Box

The Burner dialog box consists of three tabs:

- "Input/Output Tab"
- "Content Tab"
- "Command File Tab"

The Burner dialog box tabs are initialized with the values of the last burn session. Values are written to the project.ini file in the [BURNER] section.

Input/Output Tab

In the Input/Output tab, specify which file the burner uses for input and where to write the output. Click the Execute button to start the operation.

Output from the burn process usually goes to a PROM burner connected to the serial port. Output also may be redirected to a file written in either Intel Hex format, as Freescale S-Records or as plain binary.

Figure 15.2 Burner Dialog Box Input/Output Tab
Interactive Burner (GUI)
Burner Dialog Box

Input Group
Specify the input file in the Input File: text field. The browse button on the right side is used to browse for a file. The following file types are supported:

- Absolute files produced by linker. The absolute file format may be either Freescale (former Hiware) or ELF/DWARF
- S Record File
- Intel Hex File

The corresponding Batch Burner command is “SENDBYTE: Transfer Bytes” or “SENDWORD: Transfer Words”.

To specify the input file, you can use the following macro %ABS_FILE% where ABS_FILE is passed by an environment variable. See the description of environment variables.

For example:
-ENV" ABS_FILE=file_name"

Output Group
Output is written to a serial port (COM1, COM2, COM3 or COM4) or a file.

File
Select the File option button to write output to a file. In the corresponding text box, you can enter the output file name or browse for an existing file.

The corresponding Batch Burner command is “OPENFILE: Open Output File”.

If you use the macro %ABS_FILE% for the input file, you can add an extension to automatically generate the output file.

Example:
%ABS_FILE%.s19

Com1, Com2, Com3, Com4
To write the output to a serial port, select an available port and define the communication settings.

The corresponding Batch Burner command is “OPENCOM: Open Output Communication Port”.

HC(S)08/RS08 and HC(S)12 Build Tools Utilities Manual
Com Settings Group

When output is written to a serial port (COM1, COM2, COM3 or COM4), Baud Rate, Parity, Data bits and Header File are specified in the list boxes and text box of the Com Settings group.

Baud Rate

Supported Baud Rates: 300, 600, 1200, 2400, 4800, 9600, 19200 and 38400

The corresponding Batch Burner command is “\texttt{baudRate: Baudrate for Serial Communication}”.

Parity

Communication parity can be set to none, even or odd.

The corresponding Batch Burner command is “\texttt{parity: Set Communication Parity}”.

Data Bits

Number of data bits transferred can be set to 7 or 8 bits.

The corresponding Batch Burner command is “\texttt{dataBit: Number of Data Bits}”.

Header File

You can specify an initialization file for the PROM burner. This file is sent to the PROM burner byte by byte (binary) without modification before anything else is sent.

The corresponding Batch Burner command is “\texttt{header: Header File for PROM Burner}”.

Execute Group

The Execute group selects a data width and writes data. From the dropdown list select which byte or word is to be written:

- 1st Byte (msb)
- 2nd Byte
- 3rd Byte
- 4th Byte
- 1st Word
- 2nd Word

Click the Execute command button to write the data. Depending on the data width chosen, you may have to send the result to more than one output file.

Example: Format is S Record and data bus is 2 Bytes
Two output files are generated. Data for the 1st Byte (msb) is sent to a file named fibo_1.s19 and data for the 2nd byte is sent to fibo_2.s19.

Select 1st Byte (msb) and click Execute to transfer the code bytes, if you select a data bus width of 1 byte.

If your data bus is 2 bytes wide, the code is split into two parts. Selecting the 1st Byte (msb) and clicking Execute will transfer the even part of the data (corresponding to D8 to D15). Selecting 2nd Byte will transfer the odd part, which corresponds to LSB or D0 to D7.

If the data bus is 4 bytes wide, 1st Byte (msb) transfers D24 to D31, while 4th Byte sends the LSB (D0 to D7).

If using 16bit EPROMs, select one of the Word formats. If necessary, you can exchange the high and low byte. Check Swap Bytes in the Content tab of the Burner dialog box.

The corresponding Batch Burner commands are “SENDBYTE: Transfer Bytes” and “SENDWORD: Transfer Words”.
**Content Tab**

In the Burner dialog box content tab, the data format and range to be written is specified.

**Figure 15.3 Burner Dialog Box Content Tab**

![Burner Dialog Box Content Tab]

**Format Group**

Use the option boxes of the format group to select an output format. The Burner utility supports three output formats:

- S Records
- Intel Hex Files
- Binary Files

The corresponding Batch Burner command is "format: Output Format".
**Interactive Burner (GUI)**

**Burner Dialog Box**

---

**SRecord Configuration Group**
Use the option boxes and text box of the SRecord configuration group to select the type of SRecord and bytes per line to be written.

For SRecords, the type can be set to automatic, S1, S2 or S3.

The corresponding Batch Burner command is **“SRECORD: S-Record Type”**.

The number of bytes per SRecord line can be configured. This is useful when using tools with restricted capacity. The Batch Burner command is **“SLINELEN: SRecord Line Length”**.

---

**Data Bus Group**
Use the option boxes of the Data Bus group to select a Data Bus width. Data Bus/Data Width may be either 1, 2 or 4 bytes.

The corresponding Batch Burner command is **“busWidth: Data Bus Width”**.

---

**Swap Bytes Checkbox**
Use the Swap Bytes checkbox to enable swapping bytes. Swapping bytes may be enabled, if the data bus is 2 or 4 bytes.

The corresponding Batch Burner command is **“swapByte: Swap Bytes”**.

---

**Undef Byte Textbox**
For a binary output file, normally all undefined bytes in the output are written as 0xFF. If desired, another pattern can be specified.

The corresponding Batch Burner command is **“undefByte: Fill Byte for Binary Files”**.

---

**Range to Copy Group**
In the Range to Copy group, the origin (start), length, and offset is specified. The text box on the right explains the result.

Example: If your application is linked at address $3000 to $4000 and the EPROM is at address $2000 (Origin) and Length is $2000, the code will start at address $1000 relative to the EPROM. If the EPROM is at address $3000 (Origin) and Length is $1000, it is filled from the start.

---

**Origin Textbox**
Enter in this textbox has to be set to the EEPROM start address in your system.

The corresponding Batch Burner command is **“origin: EEPROM Start Address”**.

---

**Length Textbox**
Enter in this textbox is the range of program code that is actually copied.
Interactive Burner (GUI)
Burner Dialog Box

The corresponding Batch Burner command is “len: Length to be Copied”.

Destination Offset Textbox
Entry in this textbox is an additional offset which will be added to the resulting S Record or Intel Hex File. For example, if the Origin is set to 0x3000 and the Destination offset is 0x1000, then the written address will be 0x4000.

The corresponding Batch Burner command is “destination: Destination Offset”.

Command File Tab
In the Command File tab of the Burner dialog box, a summary of your settings are displayed as Batch Burner commands. You can select and copy the commands for use in make files or Batch Burner Language (.bbl) files.

Figure 15.4 Burner Dialog Box Command File Tab
If you use the selection displayed on the Command File tab in a make file, you must either place everything on a single line or use the line continuation character (\) as shown.

```
burn:
    $(BURN) \ 
    OPENFILE "fibo.s19" \ 
    format = motorola \ 
    origin = 0xE000 \ 
    len = 0x2000 \ 
    busWidth = 1
```
Batch Burner

This Chapter describes the Batch Burner Language (BBL).

Batch Burner User Interface

Starting the Burner utility displays the following window:

Figure 16.1 Burner Default Configuration Window

To use the Batch Burner, you can type in your batch burner commands on the command
line, specify a file using the -F option on the command line, or as a startup option:

-FFibobbl

or

OPENFILE "fibo.s19" origin=0xE000 len=0x2000 SENDBYTE 1 "fibo.abs"

You can also specify options and burner commands with the burner program.

burner.exe -Ffibo.bbl

You can also use the Burner directly from a make file:

burn:

$(BURN) \

$(BURN)
Batch Burner
Syntax of Burner Command Files

OPENFILE "fibo.s19" \
format = freescale \
origin = 0xE000 \
len = 0x2000 \
busWidth = 1 
SENDBYTE 1 "fibo.abs"

Syntax of Burner Command Files

The syntax of burner commands is shown in the following example.

Listing 16.1  Example of Burner Command File Syntax

StatementList = Statement {Separator Statement}.
Statement = [IfSat | ForStat | Open | Send | Close | Pause
| Echo | Format | SFormat | Origin | Len
| BusWidth | Parity | SwapByte | Header
| BaudRate | DataBit | UndefByte
| Destination | AssignExpr | SLineLen].
IfStat = "IF" RelExpr "THEN" StatementList
"ELSE" StatementList "END".
Assign = ("=" | ":=").
ForStat = "FOR" Ident Assign SimpleExpr "TO" SimpleExpr
"DO" StatementList "END".
Send = ("SENDBYTE" | "SENDWORD") SimpleExpr String.
Close = "CLOSE".
Pause = "PAUSE" [String].
Echo = "ECHO" [String].
Format = "format" Assign ("freescale" | "intel" | "binary").
SFormat = "SRECORD" Assign ("Sx" | "S1" | "S2" | "S3").
Origin = "origin" Assign SimpleExpr.
Len = "len" Assign SimpleExpr.
BusWidth = "busWidth" Assign ("1" | "2" | "4").
Parity = "parity" Assign ("none" | "even" | "odd").
SwapByte = "swapByte" Assign ("yes" | "no").
Header = "header" Assign string.
BaudRate = "baudRate" Assign ("300" | "600" | "1200"
| "2400" | "4800" | "9600"
| "19200" | "38400").
DataBit = "dataBit" Assign (7 | 8).
UndefByte = "undefByte" Assign SimpleExpr.
Destination = "destination" Assign SimpleExpr.
SLineLen = "SLINELEN" Assign SimpleExpr.
AssignExpr = Ident Assign SimpleExpr.
RelExpr = SimpleExpr {RelOp SimpleExpr}.
RelOp = "=" | "=" | "#" | "<>" | "!=" | "<" | "<=" |">" |">=".
SimpleExpr = ["+" | "-"] Term {AddOp Term}.
AddOp = "+" | "-".
Term = Number | String | Ident.
Number = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | (Number)
Ident = "i".
String = "\"" (char) "."

NOTE The identifier used in a FOR statement must be called "i".

Command File Comments
Command files accept both ANSI-C style or Modula-2 style comments.

Example
/* This is a C like comment */
(* This is a Modula-2 like comment *)

Assignments can be specified using ANSI-C or Modula-2 syntax:

Example
dataBit := 2 (* Modula-2 like *)
dataBit = 2 /* C like */

Constant format can be specified using either ANSI-C or Modula-2 syntax:

Example
origin = 0x1000 /* C like */
origin := 1000H (* Modula-2 like *)
In a makefile, the burner can be used in two different ways. The first way is to specify a command file:

\texttt{BURNER.EXE -f "<CmdFile>"}

The second way is to directly specify commands on the command line:

\texttt{BURNER.EXE SENDBYTE 1 "InFile.abs"}

If the commands are long, you can use line continuation characters in your make file as below:

\texttt{burn:}
\begin{verbatim}
$(BURN) \ \
  OPENFILE "fibo.s19" \ 
  format = freescale \ 
  origin = 0xE000 \ 
  len = 0x2000 \ 
  busWidth = 1
\end{verbatim}

If the second method is used, parameter initialization may be included in the file \texttt{DEFAULT.ENV} located in the working directory. This will reduce the length of the command line parameters, which are limited to 65535 bytes. Variables that can be specified using environment variables are listed below:

\begin{verbatim}
header= \ 
format=freescale \ 
busWidth=1 \ 
origin=0 \ 
len=0x1000 \ 
parity=none \ 
undefByte=0xff \ 
baudRate=9600  
dataBit=8  
swapByte=no
\end{verbatim}

The example above shows the default values but any legal value can be assigned (see section \textit{Parameters of the Command File}). For further details, see the example in the following section.
Command File Examples

The examples below show how to write a command file. Listing 16.2 shows a command file for conditional and repetitive statements.

If the symbol # appears in a string it is replaced by the value of i.

Listing 16.2 Sample Command File for Conditional and Repetitive Statements

```
ECHO
ECHO "I can count... and I can take decisions"
FOR i = 0 TO 8 DO
  IF i == 7 THEN
    ECHO "This is the number seven"
  ELSE
    ECHO "#"
  END
  IF i == 3 THEN
    ECHO "This was the number three"
  END
END
```

Listing 16.3 shows a command file for redirecting the output to a file. To redirect output, use the command OPENFILE.

Listing 16.3 Command File for Redirecting Output

```
ECHO
ECHO "Programming 2 EPROMs with 3 files"
ECHO "the first byte of the word goes into the first EPROM"
ECHO "the second byte of the word goes into the second EPROM"
PAUSE "Hit any key to continue"
  format = freescale
  busWidth = 2
  origin = 0
  len = 0x3000
FOR i = 1 TO 2 DO
  PAUSE "Insert EPROM n# and press <return>"
  OPENFILE "prom#.bin"
  origin = 0X
  SENDBYTE i "demo1.abs"
  origin = origin + 0x500
  SENDBYTE i "demo2.abs"
  origin = origin + 0x500
  SENDBYTE i "demo3.abs"
  CLOSE
END
```
Batch Burner
Batch Burner with Makefile

Listing 16.4 shows a command file for redirecting the output to a serial port. To redirect output to serial port, use the command OPENCOM.

Listing 16.4 Command File for Redirecting Output to Serial Port

```
ECHO
ECHO "I can also program 16-bit EPROMs with header"
PAUSE "Hit any key to continue"
header = "init.prm"
format = intel
busWidth = 2
origin = 0x0
len = 0x1000
OPENCOM 1 /* here com1, com2, com3 or com4 could be used*/
SENDWORD 1 "fbin1.map"
CLOSE
```

Listing 16.5 shows a command file for calling the burner from a makefile. After compiling and linking the application, the generated code is prepared to be burned into two EPROMs, one containing the odd bytes (fibo_odd.s1) and the other the even bytes (fibo_eve.s1).

Listing 16.5 Command File for Calling Burner from makefile

```
makeall:
  $(COMP) $(FLAGS) fibo.c
  $(LINK) fibo.prm
  burner.exe OPENFILE "fibo_odd.s1" \
    busWidth=2 SENDBYTE 1 "fibo.abs"
  burner.exe OPENFILE "fibo_eve.s1" \
    busWidth=2 SENDBYTE 2 "fibo.abs"
```

Note that for all parameters that are not specified in the parameter list, default values or the values specified by environment variables will be used.
Parameters of the Command File

This section describes valid parameter values that can be used in commands. For more details about commands, refer to the file FIBO.BBL, which shows how to write a script.

Following commands are available:

- **baudRate**: Baudrate for Serial Communication
- **busWidth**: Data Bus Width
- **CLOSE**: Close Open File or Communication Port
- **dataBit**: Number of Data Bits
- **destination**: Destination Offset
- **DO**: For Loop Statement List
- **ECHO**: Echo String onto Output Window
- **ELSE**: Else Part of If Condition
- **END**: For Loop End or If End
- **FOR**: For Loop
- **format**: Output Format
- **header**: Header File for PROM Burner
- **IF**: If Condition
- **len**: Length to be Copied
- **OPENCOM**: Open Output Communication Port
- **OPENFILE**: Open Output File
- **origin**: EEPROM Start Address
- **parity**: Set Communication Parity
- **PAUSE**: Wait until Key Pressed
- **SENDBYTE**: Transfer Bytes
- **SENDWORD**: Transfer Words
- **SLINELEN**: SRecord Line Length
- **SRECORD**: S-Record Type
- **swapByte**: Swap Bytes
- **THEN**: Statementlist for If Condition
- **TO**: For Loop End Condition
- **undefByte**: Fill Byte for Binary Files
Batch Burner
Parameters of the Command File

**baudRate: Baudrate for Serial Communication**

**Syntax:**

```
"baudRate" assign <baud>.
```

**Arguments:**

- `<baud>`: valid baudrate.

**Default:**

```
baudrate = 9600
```

**Description:**

Sets the transmission speed. This parameter must not be used when the burner output is redirected to a file. Valid identifier values are 300, 600, 1200, 2400, 4800, 9600, 19200 or 38400 (default is 9600).

This command is only used if output is sent to a communication port.

**Example:**

```
baudRate = 19200
```

**See also:**

- [dataBit: Number of Data Bits](#)
- [parity: Set Communication Parity](#)
- [header: Header File for PROM Burner](#)
- [OPENCOM: Open Output Communication Port](#)

---

**busWidth: Data Bus Width**

**Syntax:**

```
"busWidth" assign ("1" | "2" | "4").
```

**Arguments:**

- A bus width of 1, 2 or 4
Default:

    busWidth = 1

Description:

Most EPROMs are 1 byte wide. In order to burn an application into EPROMs, 1, 2 or 4 EPROMs are needed depending on the width of the data bus of the target system used. The Burner program allows you to select the data bus width using the identifier busWidth. Only 1, 2 and 4 are valid values for the parameter busWidth (the default is 1).

Example:

    busWidth = 4

CLOSE: Close Open File or Communication Port

Syntax:

    "CLOSE".

Arguments:

    None.

Default:

    None.

Description:

To close a file opened by OPENFILE: Open Output File or COM port opened with OPENCOM: Open Output Communication Port.

Example:

    CLOSE

See also:

- OPENFILE: Open Output File
- OPENCOM: Open Output Communication Port
**dataBit: Number of Data Bits**

**Syntax:**
```
"dataBit" assign ("7" | "8").
```

**Arguments:**
7 or 8 data bits.

**Default:**
`dataBit = 8`

**Description:**
Sets the number of data bits. This parameter must not be used when the burner output is redirected to a file. Valid identifier values are 7 or 8 (default is 8).

This command is only used if the output is sent to a communication port.

**Example:**
```
dataBit = 7
```

**See also:**
- baudRate: Baudrate for Serial Communication
- parity: Set Communication Parity
- header: Header File for PROM Burner
- OPENCOM: Open Output Communication Port

**destination: Destination Offset**

**Syntax:**
```
"destination" assign <offset>.
```

**Arguments:**
`<offset>`: offset to be added
Default:

```
destination = 0
```

Description:

With this command an additional offset may be added to the address field of an S-Record or a Intel Hex Record.

Example:

```
destination = 0x2000
```

See also:

- `len`: Length to be Copied
- `origin`: EEPROM Start Address

---

**DO: For Loop Statement List**

Syntax:

```
"FOR" Ident Assign SimpleExpr
"TO" SimpleExpr "DO" StatementList "END".
```

Arguments:

None.

Default:

None.

Description:

This command starts the FOR statement list. As ident only ‘i’ may be used, and each occurrence of # in the loop is replaced with the actual value of ‘i’.

Example:

```
FOR i=0 TO 10 DO
  ECHO "#"
END
```
Batch Burner
Parameters of the Command File

See also:
- FOR: For Loop
- TO: For Loop End Condition
- END: For Loop End or If End

ECHO: Echo String onto Output Window

Syntax:

```plaintext
"ECHO" [<string>].
```

Arguments:

- `<string>`: a string written to the output window

Default:

None.

Description:

With this command, a string can be written to the output window. If no string is specified, an empty line is written.

Example:

```plaintext
ECHO
ECHO "hello world!"
```

ELSE: Else Part of If Condition

Syntax:

```plaintext
"IF" RelExpr "THEN" StatementList
["ELSE" StatementList] "END".
```

Arguments:

None.
**Default:**
None.

**Description:**
This command starts the optional ELSE part of an IF conditional section.

**Example:**
```
FOR i=0 TO 10 DO
 IF i==7 THEN
   ECHO "i is 7"
 ELSE
   ECHO "#"
 END
 END
```

**See also:**
- [END: For Loop End or If End](#)
- [IF: If Condition](#)
- [THEN: Statementlist for If Condition](#)

---

**END: For Loop End or If End**

**Syntax:**
```
"FOR" Ident Assign SimpleExpr
 "TO" SimpleExpr "DO" StatementList "END".
```

or
```
"IF" RelExpr "THEN" StatementList
 ["ELSE" StatementList] "END".
```

**Arguments:**
None.

**Default:**
None.
**Description:**
This command ends either a FOR loop or IF condition.

**Example:**

```plaintext
FOR i=0 TO 10 DO
  IF i==7 THEN
    ECHO "i is 7"
  END
  ECHO "#"
END
```

**See also:**
- IF: If Condition
- THEN: Statementlist for If Condition
- ELSE: Else Part of If Condition
- TO: For Loop End Condition
- DO: For Loop Statement List
- FOR: For Loop

**FOR: For Loop**

**Syntax:**

```
"FOR" Ident Assign SimpleExpr
"TO" SimpleExpr "DO" StatementList "END".
```

**Arguments:**
None.

**Default:**
None.

**Description:**
This command starts a FOR loop.
Example:

```
FOR i=0 TO 10 DO
  IF i==7 THEN
    ECHO "i is 7"
  END
  ECHO "#"
END
```

See also:
- TO: For Loop End Condition
- DO: For Loop Statement List
- END: For Loop End or If End

format: Output Format

Syntax:
```
"format" assign ("freescale" | "intel" | "binary").
```

Arguments:
Format, either Freescale S, Intel Hex or Binary.

Default:
```
format = freescale
```

Description:
The Burner supports three different data transfer formats: S-Records, Intel Hex-Format and binary format. With the binary format the output destination must be a file. Valid identifiers are: Freescale, intel, binary (the default is Freescale)

Example:
```
format = binary
```
Batch Burner
Parameters of the Command File

header: Header File for PROM Burner

Syntax:
"header" assign <fileName>.

Arguments:
<fileName>: header file to be sent to serial port

Default:
header =

Description:
Specifies an initialization file for the PROM burner. This parameter must not be used when the burner output is redirected to a file. This file is sent byte by byte (binary) without modification to the PROM burner before anything else is sent. This command is only used if the output is sent to a communication port.

Example:
header = "myheader.txt"

See also:
- baudRate: Baudrate for Serial Communication
- parity: Set Communication Parity
- header: Header File for PROM Burner
- dataBit: Number of Data Bits
- OPENCOM: Open Output Communication Port

IF: If Condition

Syntax:
"IF" RelExpr "THEN" StatementList
["ELSE" StatementList] "END".
Arguments:
None.

Default:
None.

Description:
This command starts an IF conditional section.

Example:
```batch
FOR i=0 TO 10 DO
  IF i==7 THEN
    ECHO "i is 7"
  END
  ECHO "#"
END
```

See also:
- END: For Loop End or If End
- THEN: Statementlist for If Condition
- ELSE: Else Part of If Condition

---

len: Length to be Copied

Syntax:
```
"len" assign <number>.
```

Arguments:

<number>: length to be copied.

Default:

len = 0x10000
**Batch Burner**  
*Parameters of the Command File*

**Description:**
Range of program code to be copied. Length can also be specified using the ANSI-C or Modula-2 notation for hexadecimal constants (default is 0x10000).

**Example:**
If an application is linked between address $3000 and $4000 and the EEPROM start address is $2000 (origin), then \( len \) must be set to $2000. The code is stored at address $1000 relative to the EEPROM start address.

If the EPROM start address is $3000 (origin) then \( len \) must be set to $1000. The code is stored at the beginning of the EPROM.

**Example:**
\[
len = 0x2000
\]

**See also:**
- destination: Destination Offset
- origin: EEPROM Start Address

---

**OPENCOM: Open Output Communication Port**

**Syntax:**
```
"OPENCOM" <port>.
```

**Arguments:**
- `<port>`: valid COM port number (1, 2, 3, 4).

**Default:**
None.

**Description:**
With this command, the Burner will send the output to the specified communication port. To close the port opened, **CLOSE: Close Open File or Communication Port** has to be used.

**Example:**
```
OPENCOM 2
```
See also:

- **baudRate**: Baudrate for Serial Communication
- **parity**: Set Communication Parity
- **header**: Header File for PROM Burner
- **dataBit**: Number of Data Bits
- **OPENFILE**: Open Output File
- **CLOSE**: Close Open File or Communication Port

---

**OPENFILE**: Open Output File

**Syntax:**

```
"OPENFILE" <file>.
```

**Arguments:**

- `<file>`: valid file name.

**Default:**

None.

**Description:**

With this command, the Burner will send the output to the specified file. To close the file, use **CLOSE: Close Open File or Communication Port** command.

**Example:**

```
OPENFILE "myFile.s19"
```

**See also:**

- **OPENCOM**: Open Output Communication Port
- **CLOSE**: Close Open File or Communication Port
origin: EEPROM Start Address

Syntax:

"origin" assign <address>.

Arguments:

<address>: start address.

Default:

origin = 0

Description:

Initialized with the EPROM start address in the target system. The start address can be specified using ANSI C or Modula-2 notation for hexadecimal constants (default is 0).

Example:

origin = 0xC000

See also:

- len: Length to be Copied
- destination: Destination Offset

parity: Set Communication Parity

Syntax:

"parity" assign ("none" | "even" | "odd").

Arguments:

parity none, even or odd.

Default:

parity = none
Description:
Sets the parity used for transfer. This parameter must not be used when the burner output is redirected to a file. Valid identifier values are none, odd, and even (default is none). This command is only used if the output is sent to a communication port.

Example:
parity = even

See also:
- baudRate: Baudrate for Serial Communication
- dataBit: Number of Data Bits
- header: Header File for PROM Burner
- OPENCOM: Open Output Communication Port

SENDBYTE: Transfer Bytes

Syntax:
“SENDBYTE” <number> <file>.

Arguments:
<number>: valid byte number (1, 2, 3, 4)
<file>: valid source file name.

Default:
None.

Description:
This command starts the transfer.
If the data format is binary, the destination must be a file. The size of the file is the size specified by len divided by the busWidth. All undefined bytes are initialized with $FF or with the value specified by undefByte: Fill Byte for Binary Files.
If a data bus width of 1 byte is selected, the following command must be used to transfer the code:
SENDBYTE 1 "InFile.abs"
If the data bus is 2 bytes wide, the code is split into two parts; the command `SENDBYTE 1 "InFile.abs"` transfers the even part of the code (corresponding to D8 to D15) while the command `SENDBYTE 2 "InFile.abs"` transfers the odd part, which corresponds to the LSB (D0 to D7).

If the data bus is 4 bytes wide, the command `SENDBYTE 1 "InFile.abs"` transfers the MSB (D24 to D31), while the command `SENDBYTE 4 "InFile.abs"` sends the LSB (D0 to D7).

Using 16-bit EPROMs the commands `SENDWORD 1 "InFile.abs"` and `SENDWORD 2 "InFile.abs"` must be used. If necessary, high and low byte can be swapped by initializing swapBytes with yes.

**Example:**

```
SENDBYTE 1 "myApp.abs"
```

**See also:**
- `busWidth: Data Bus Width`
- `SENDWORD: Transfer Words`

---

### SENDWORD: Transfer Words

**Syntax:**

```
"SENDWORD" <number> <file>.
```

**Arguments:**

- `<number>`: valid word number (1, 2)
- `<file>`: valid source file name.

**Default:**

None.

**Description:**

This command starts the transfer.

If the data format is binary, the destination must be a file. The size of the file is the size specified by `len` divided by the `busWidth`. All undefined bytes are initialized with $FF$ or value specified by `undefByte: Fill Byte for Binary Files`. 
If a data bus width of 1 byte is selected, the following command must be used to transfer the code:

```
SENDBYTE: Transfer Bytes 1 “InFile.abs”
```

If the data bus is 2 bytes wide, the code is split into two parts; the command `SENDBYTE 1 “InFile.abs”` transfers the even part of the code (corresponding to D8 to D15) while the command `SENDBYTE 2 “InFile.abs”` transfers the odd part, which corresponds to the LSB (D0 to D7).

If the data bus is 4 bytes wide, the command `SENDBYTE 1 “InFile.abs”` transfers the MSB (D24 to D31), while the command `SENDBYTE 4 “InFile.abs”` sends the LSB (D0 to D7).

Using 16-bit EPROMs, the commands `SENDWORD 1 “InFile.abs”` and `SENDWORD 2 “InFile.abs”` must be used. If necessary, the high and low byte can be swapped by initializing swapBytes with yes.

**Example:**

```
SENDWORD 1 “myApp.abs”
```

**See also:**
- `SENDBYTE: Transfer Bytes`
- `busWidth: Data Bus Width`

---

### SLINELEN: SRecord Line Length

**Syntax:**

```
“SLINELEN” assign <number>.
```

**Arguments:**

- `<number>`: valid line length (1, 2,...)

**Default:**

```
<number> == 32.
```

**Description:**

This command configures how many bytes written are on a single SRECORD line. This command only effects SRECORD file generation.
Example:

With SLINELEN 16, the burner generates:
\begin{verbatim}
S11320000000000010100000000000000CA
S1132010008800208208000000001020408106B
\end{verbatim}

With SLINELEN 8, the burner generates:
\begin{verbatim}
S10B2000000000001010000D2
S10B200800000000000000CC
S10B20100088002082080092
S10B20180000001020408109D
\end{verbatim}

See also:
- format: Output Format

SRECORD: S-Record Type

Syntax:

```
"SRECORD=" ("Sx" | "S1" | "S2" | "S3").
```

Arguments:

- **"Sx"**: Automatic choose between S1, S2 or S3 records
- **"S1"**: use S1 records
- **"S2"**: use S2 records
- **"S3"**: use S3 records

Default:

```
SRECORD=Sx.
```

Description:

This command is for S-Record output format.

Normally the Burner chooses the matching S Record type depending on the addresses used. However, with this option a certain type may be forced because the PROM burner only supports one type.
If $Sx$ is active, the burner is in automatic mode:
if the highest address is $\geq 0x1000000$, then $S3$ records are used,
if the highest address is $\geq 0x10000$, then $S2$ records are used,
otherwise $S1$ records are used.

**Example:**

```
SRECORD=S2
```

**See also:**

- [format: Output Format](#)

---

**swapByte: Swap Bytes**

**Syntax:**

```
"swapByte" assign ("on" | "off").
```

**Arguments:**

- "on": enables byte swapping
- "off": disables byte swapping

**Default:**

```
swapByte = off
```

**Description:**
If necessary, the high and low byte can be exchanged when 16-bit or 32-bit EPROMs are used.

**Example:**

```
swapByte = on
```

**See also:**

- [busWidth: Data Bus Width](#)
THEN: Statementlist for If Condition

Syntax:

```
"IF" RelExpr "THEN" StatementList
["ELSE" StatementList] "END".
```

Arguments:

None.

Default:

None.

Description:

This command starts an IF conditional section.

Example:

```
FOR i=0 TO 10 DO
  IF i==7 THEN
    ECHO "i is 7"
  END
  ECHO "#"
END
```

See also:

- END: For Loop End or If End
- IF: If Condition
- ELSE: Else Part of If Condition
TO: For Loop End Condition

Syntax:

"FOR" Ident Assign SimpleExpr
"TO" SimpleExpr "DO" StatementList "END".

Arguments:
None

Default:
None

Description:
Specifies the FOR loop end condition. As ident, only ‘i’ may be used, and each occurrence of # in the loop is replaced with the actual value of ‘i’.

Example:

FOR i=0 TO 10 DO
  ECHO "#"
END

See also:
- FOR: For Loop
- DO: For Loop Statement List
- END: For Loop End or If End
**undefinedByte: Fill Byte for Binary Files**

**Syntax:**

```
"undefinedByte" assign <number>.
```

**Arguments:**

- `<number>`: 8bit number

**Default:**

`undefinedByte = 0xFF`

**Description:**

This command assigns the default fill byte to undefined bytes in binary output files. This command is only used for binary files.

**Example:**

`undefinedByte = 0x33`

**See also:**

- `format: Output Format`

---

**PAUSE: Wait until Key Pressed**

**Syntax:**

```
"PAUSE" [<string>]
```

**Arguments:**

- `<string>`: a string written to output window

**Default:**

None

**Description:**

This command causes the batch burner language program to wait until a key is pressed. An optional message text may be specified. For Windows 95/98/NT, a dialog box appears:
Figure 16.2 Burner Pause Dialog Box

Example:

PAUSE "please press a key."
Burner Options

The Burner utility offers a number of options that you can use to control the application. Options are composed of a minus sign/dash ('-') followed by one or more letters or digits. Command line options are not case sensitive, e.g. "-F" is the same as "-f".

The burner command line can contain the name of a file to be built with the -F: Execute Command File, or a list of commands.

Options before the first command on the command line are recognized. Then, all remaining text is taken as arguments to the command, including options.

For example:

`OPENFILE "fibo.out" format=freescale len=0x1000 SENDBYTE l "fibo.abs.abs" CLOSE`  
Command is executed.

-f=fibo.bbl  
Command file fibo.bbl is executed.

-f fibo.bbl  
This is an alternate form of the recommended "-f=fibo.bbl". This form is allowed for compatibility only.

The following is not allowed:

fibo.bbl -f  
fibo.bbl is interpreted as a command with argument -f. This generates an error, since no such command exists.
Burner Option Details

Option Groups

Options are grouped into four categories: HOST, INPUT, MESSAGE and VARIOUS. The group corresponds to the property sheets of the graphical option settings.

<table>
<thead>
<tr>
<th>Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOST</td>
<td>Lists options related to host.</td>
</tr>
<tr>
<td>INPUT</td>
<td>Lists options related to input file.</td>
</tr>
<tr>
<td>MESSAGES</td>
<td>Lists options that generate error messages.</td>
</tr>
<tr>
<td>VARIOUS</td>
<td>Lists various options.</td>
</tr>
</tbody>
</table>

NOTE Not all command line options are accessible through the property sheets, for example, -H or -Lic).

Option Detail Description

The remainder of this section describes options available for the application. Options are listed in alphabetical order and divided into several sections.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td>HOST, INPUT, MESSAGE or VARIOUS.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Specifies syntax of option in EBNF format.</td>
</tr>
<tr>
<td>Arguments</td>
<td>Describes and lists optional and required arguments for the option.</td>
</tr>
<tr>
<td>Default</td>
<td>Shows default setting for option.</td>
</tr>
<tr>
<td>Description</td>
<td>Provides a detailed description of the option and how to use it.</td>
</tr>
<tr>
<td>Example</td>
<td>Gives an example of usage, and effects of the option where possible.</td>
</tr>
<tr>
<td>See also</td>
<td>Names related topics.</td>
</tr>
</tbody>
</table>
-D: Display Dialog Box

Group:
VARIOUS

Syntax:
"-D".

Arguments:
None.

Default:
None.

Description:
This option displays the Burner dialog box. This interface, with its three tabs, allows you to launch the burner from a make file and await user input.

Figure 17.1 Burner Dialog Window Input/Output Tab

Example:
burner.exe -D
Burner Options

-Env: Set Environment Variable

Group:
    HOST

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

Arguments:
    <Environment Variable>: Environment variable to be set
    <Variable Setting>: Value to be assigned

Default:
    None.

Description:
    This option sets an environment variable. This environment variable may be used in the
    maker or used to overwrite system environment variables.

Example:
    -EnvOBJPATH=\sources\obj
    This is the same as
    OBJPATH=\sources\obj
    in the default.env file

-F: Execute Command File

Group:
    INPUT

Syntax:
    "-F=" <fileName>.
Arguments:

`<fileName>`: Batch Burner command file to be executed.

Default:
None.

Description:
This option causes the Burner to execute a Batch Burner command file (usual extension is `.bbl`).

Example:

```
-F=fibo.bbl
```

-H: Short Help

Group:

VARIOUS

Syntax:

"-H".

Arguments:
None.

Default:
None.

Description:
The `-H` option displays a short list (i.e. help list) of available options within main window.
No other option or source file should be specified with the `-H` option.
Burner Options

Example:

-H may produce following list:

... VARIOUS:
-H Prints this list of options
-V Prints the Compiler version
...

-Lic: License Information

Group:
VARIOUS

Syntax:
"-Lic".

Arguments:
None.

Default:
None.

Description:
The -Lic option prints the current license information (e.g. if it is a demo version or a full version). This information is also displayed in the about box.

Example:
-Lic

See also:
- "-LicA: License Information about Every Feature in Directory"
- "-LicBorrow: Borrow License Feature"
- "-LicWait: Wait for Floating License from Floating License Server"
-LicA: License Information about Every Feature in Directory

Group:
    VARIOUS

Syntax:
    "-LicA".

Arguments:
    None.

Default:
    None.

Description:
The -LicA option prints the license information of every tool or .dll in the directory containing the executable (for example, license information can indicate whether a tool or feature is a demo version or full version). Because the option analyzes every file in the directory, this may take a long time.

Example:
    -LicA

See also:
    - "-Lic: License Information"
    - "-LicBorrow: Borrow License Feature"
    - "-LicWait: Wait for Floating License from Floating License Server"

-LicBorrow: Borrow License Feature

Group:
    HOST

Syntax:
    "-LicBorrow"<feature>[";"<version>]"<Date>.
Burner Options

Arguments:

<feature>: the feature name to be borrowed (e.g. HI100100).
<version>: optional version of the feature to be borrowed (e.g. 3.000).
<date>: date with optional time until when the feature shall be borrowed (e.g. 15-Mar-2004:18:35).

Default:

None.

Description:

This option allows you to borrow a license feature until a given date/time. Borrowing allows you to use a floating license even if disconnected from the floating license server.

You need to specify the feature name and the date until you want to borrow the feature. If the feature you want to borrow is a feature belonging to the tool where you use this option, then you do not need to specify the version of the feature (because the tool knows the version). However, if you want to borrow any feature, you need to specify its feature version as well.

You can check the status of currently borrowed features in the tool about box.

You only can borrow features, if you have a floating license and if your floating license is enabled for borrowing. See as well the provided FLEXlm documentation about details on borrowing.

Example:

-LicBorrowHI100100;3.000:12-Mar-2004:18:25

See also:

- “-Lic: License Information”
- “-LicA: License Information about Every Feature in Directory”
- “-LicWait: Wait for Floating License from Floating License Server”
-LicWait: Wait for Floating License from Floating License Server

Group:
HOST

Syntax:
"-LicWait".

Arguments:
None.

Default:
None.

Description:
By default, if a license is not available from the floating license server, then the application will immediately return. With -LicWait set, the application will wait (blocking) until a license is available from the floating license server.

Example:
-LicWait

See also:
- "-Lic: License Information"
- "-LicA: License Information about Every Feature in Directory"
- "-LicBorrow: Borrow License Feature"
Burner Options

-N: Display Notify Box

Group:
  MESSAGE

Syntax:
  "-N".

Arguments:
  None.

Default:
  None.

Description:
Causes the application to display an alert box if there was an error during the process. This is useful when running a make file (refer to Make Utility) since the tool waits for the user to acknowledge the message, thus suspending make file processing. This option is only present in PC versions.

Example:
  -Fnofile -N
  
  If an error occurs during processing, a dialog box similar to the following one will appear:

Figure 17.2 Burner Error Dialog Box

![Burner Error Dialog Box](image)
-NoBeep: No Beep in Case of an Error

Group:
MESSAGE

Syntax:
"-NoBeep".

Arguments:
None.

Default:
None.

Description:
Normally there is a ‘beep’ at the end of processing, if an error occurs. Use this option to turn off the beep.

Example:
-NoBeep

-NoEnv: Do Not Use Environment

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

Syntax:
"-NoEnv".

Arguments:
None.

Default:
None.
Burner Options

**Description:**
This option can only be specified on the command line, while starting the application. It cannot be specified in an environment file, such as the `default.env` file.

When this option is used, the application does not use an environment file, such as `default.env`, `project.ini` or `tips` file.

**Example:**
```
burner.exe -NoEnv
```

**See also:**
- “Local Configuration File (Usually project.ini)”

-Ns: Configure S-Records

**Group:**
OUTPUT

**Syntax:**
```
--Ns "=" \("p" | "0" | "7" | "8" | "9"") \.
```

**Arguments:**
- "p": no path in S0 record
- "0": no S0 record
- "7": no S7 record
- "8": no S8 record
- "9": no S9 record

**Default:**
None.

**Description:**
Usually an S-Record file contains a S0-Record at the beginning that contains the name of the file and an S7, S8 or S9 record at the end, depending on the address size. For the S3 format, an S7 record is written at the end. For S2 format, an S8 record is written at the end. For the S1 format, an S9 record is written at the end.

This feature is useful for disabling some S-Record generation in case a non-standard S-Record file reader cannot read S0, S7, S8 or S9 records.
Burner Options

In case the option is specified without suboptions (only \(-Ns\)), no start (S0) and no end records (S7, S8 or S9) are generated.

The option \(-Ns=p\) removes the path (if present) from the file name in the S0 record.

**Example:**

\[-Ns=0\]

**See also:**

- “[SRECORD: S-Record Type](#)”.

---

**-Prod: Specify Project File at Startup**

**Group:**

None. (this option can not be specified interactively)

**Syntax:**

“-Prod=\<file>”.

**Arguments:**

\<file\>: name of project or project directory

**Default:**

None.

**Description:**

This option can only be specified on the command line while starting the application. It can not be specified in any other situation, including the default.env file.

When this option is used, the application opens the specified file as a configuration file. If a directory is specified, the default name `project.ini` is appended. If loading fails, a message box appears.

**Example:**

`burner.exe -prod=project.ini`

**See also:**

- “[Local Configuration File (Usually project.ini)](#)”
Burner Options

-V: Prints Version Information

Group:
   VARIOUS

Syntax:
   "-V".

Arguments:
   None.

Default:
   None.

Description:
Prints the application version and versions of modules internal to the application, and the current directory.

NOTE  This option is useful to determine the current directory of the application. This option is NOT present in the dialog box.

Example:
-V produces the following list:
Directory: \software\sources
Common Module V-5.0.7, Date Apr 12 1999
User Interface Module, V-5.0.18, Date Apr 8 1999

-View: Application Standard Occurrence

Group:
   HOST

Syntax:
   "-View" <kind>.
Arguments:

\(<kind>\) is one of:

"Window": Default window size
"Min": Application window is minimized
"Max": Application window is maximized
"Hidden": Application window is not visible (only if arguments)

Default:

If the application started with arguments: Minimized.
If the application started without arguments: Window.

Description:

The application (e.g. linker, compiler,...) is started with a default 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 minimized to allow batch processing. However, with this option the window mode may be specified.

Use \(-\text{ViewWindow}\) to display application in its normal window. Use \(-\text{ViewMin}\) to start application as an icon in the task bar.
Use \(-\text{ViewMax}\) to start application window maximized.
Use \(-\text{ViewHidden}\) for the application to process arguments (e.g. files to be compiled/linked) in the background (no window/icon visible). If you use the -N option, a dialog box is still possible.

NOTE This option is only present on the IBM PC version.

Example:

\(c:\Freescale\prog\burner.exe \text{-ViewHidden fibo.bbl}\)

-W: Display Window

Group:

VARIOUS

Syntax:

"-W".
Burner Options

Arguments:
None.

Default:
None.

Description:
In the V2.7 Burner, this option was used to show the Batch Burner Window. This option is ignored with the V5.x versions or later.

NOTE
This option is only provided for compatibility reasons, and is NOT present in the dialog box.

Example:
burner.exe -W

-Wmsg8x3: Cut File Names in Microsoft Format to 8.3

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, which means the file name can have a maximum of 8 characters with a maximum 3 character extension. Longer file names are possible with Win95 or WinNT.

This option causes the file name in the Microsoft message to be truncated to the 8.3 format.
Example:

x:\mysourcefile.bbl(3): INFORMATION B1000: message text

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

x:\mysource.bbl(3): INFORMATION C2901: message text

See also:

• "-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode"
• "-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode"

-WErrFile: Create “err.log” Error File

Group:

MESSAGE

Syntax:

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

Arguments:

None.

Default:

err.log is created/deleted.

Description:

A return code provides error feedback from the application to called tools. In a 16 bit windows environment, this was not possible, so a file "err.log" was used to store errors. To state no error, the file "err.log" was deleted. With UNIX or WIN32 applications, a return code is available, so the error file is no longer needed. To use a 16 bit maker with this tool, the error file must be created in order to signal an error.

Example:

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

-WErrFileOff
existing err.log is not modified.
**Burner Options**

---

**See also:**
- “-WStdout: Write to Standard Output”
- “-WOutFile: Create Error Listing File”

---

**-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:**

This option is used to change the error message color. The value is specified as a RGB (Red-Green-Blue) value in decimal format. Hexadecimal needs a 0X prefix, (for gray errors: -WmsgCE0x808080).

**Example:**

```
-WmsgCE255 changes error messages to blue.
```
-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:
This option specifies the fatal message color. The value is specified as an RGB (Red-Green-Blue) value in decimal format. Hexadecimal needs a 0X prefix, (for gray errors: -WmsgCF0x808080).

Example:
-WmsgCF255 changes fatal messages to blue.

-WmsgCI: RGB Color for Information Messages

Group:
MESSAGE

Scope:
Function
Burner Options

Syntax:

"-WmsgCI" <RGB>.

Arguments:

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

Default:

-WmsgCI32768 (r00 g80 b00, green)

Defines:

None.

Description:

This option specifies the information message color. The value is specified as an RGB (Red-Green-Blue) value in decimal format. Hexadecimal needs a 0X prefix, (for gray errors: -WmsgCI0x808080).

Example:

-WmsgCI255 changes information messages to blue.

-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)
Burner Options

Defines:
None.

Description:
This option specifies the user message color. The value is specified as an RGB (Red-Green-Blue) value in decimal format. Hexadecimal needs a 0X prefix, (for gray errors: -WmsgCU0x808080).

Example:
- WmsgCU255 changes user messages to blue.

-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:
This option specifies the warning message color. The value is specified as an RGB (Red-Green-Blue) value in decimal format. Hexadecimal needs a 0X prefix, (for gray errors: -WmsgCW0x808080).
Burner Options

Example:

-WmsgCW0 changes warning messages to black.

-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode

Group:
MESSAGE

Syntax:
“-WmsgFb“ [“v” | “m”].

Arguments:
“v”: Verbose format.
“m”: Microsoft format.

Default:
-WmsgFbm

Description:
The Application can be started with additional arguments (e.g. files to be processed together with options). If the application has been started with arguments (e.g. from the Make Tool or with the ‘%f’ argument from WinEdit), it processes the files in a batch mode. No application window is visible and the application terminates after job completion.

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

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

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 verbose message format may slow down processing, because the tool has to write more information into the message file.
**Example:**

By default, the application may produce following file if it is running in batch mode (e.g. started from the Make tool):

```
X:\C.bbl(3): INFORMATION B2901: Message
```

Setting the format to verbose stores more information in the file:

```
-WmsgFbw
```

```
>> in "X:\C.bbl", line 3, col 2, pos 33
   some text
   ^
INFORMATION B2901: Message
```

**See also:**

- “ERRORFILE: Error File Name Specification”
- “-WmsgFi (-WmsgFi, -WmsgFim): Set Message Format for Interactive Mode”

**-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode**

**Group:**

MESSAGE

**Syntax:**

```
"-WmsgFi{ "v" | "m" }.
```

**Arguments:**

“v”: Verbose format.

“m”: Microsoft format.

**Default:**

```
-WmsgFiv
```
Burner Options

Description:
If the application is started without additional arguments (e.g. files to be processed and options), it is in interactive mode (window is visible).

By default, the application uses the verbose error file format to write 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).

NOTE Using the Microsoft format may speed up processing, since the application has to write less information to the screen.

Example:
By default, the application may produce the following error output in the application window if it is running in interactive mode:

```
Top: X:\C.bbl

>> in “X:\C.bbl”, line 3, col 2, pos 33
   some text
^
INFORMATION B2901: Message
```

Setting the format to Microsoft, less information is displayed:

```
-WmsgFim
Top: X:\C.bbl
X:\C.bbl(3): INFORMATION B2901: Message
```

See also:
- “ERRORFILE: Error File Name Specification”
- “-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode”
-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 to modify the default message format in batch mode. The following formats are supported (example source file x:\Freescale\mysourcefile.cpph):

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%s</td>
<td>Source Extract</td>
<td>x:\Freescale\</td>
</tr>
<tr>
<td>%p</td>
<td>Path</td>
<td>x:\Freescale\mysourcefile</td>
</tr>
<tr>
<td>%f</td>
<td>Path and name</td>
<td>mysourcefile</td>
</tr>
<tr>
<td>%n</td>
<td>File name</td>
<td>.cpph</td>
</tr>
<tr>
<td>%N</td>
<td>File ($ chars)</td>
<td>mysource</td>
</tr>
<tr>
<td>%E</td>
<td>Extension (3 chars)</td>
<td>.cpp</td>
</tr>
<tr>
<td>%l</td>
<td>Line</td>
<td>3</td>
</tr>
<tr>
<td>%c</td>
<td>Column</td>
<td>47</td>
</tr>
<tr>
<td>%o</td>
<td>Pos</td>
<td>1234</td>
</tr>
<tr>
<td>%K</td>
<td>Uppercase kind</td>
<td>ERROR</td>
</tr>
<tr>
<td>%k</td>
<td>Lowercase kind</td>
<td>error</td>
</tr>
<tr>
<td>%d</td>
<td>Number</td>
<td>B1815</td>
</tr>
<tr>
<td>%m</td>
<td>Message</td>
<td>text</td>
</tr>
<tr>
<td>%%</td>
<td>Percent</td>
<td>%</td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
<td></td>
</tr>
</tbody>
</table>
Burner Options

Example:

-\WmsgFoi"%f%e(%l): %k %d: %m"

produces a message in following format:
X:\C.C(3): information B2901: Message

See also:
- “ERRORFILE: Error File Name Specification”
- “-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode”
- “-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode”
- “-WmsgFonp: Message Format for No Position Information”
- “-WmsgFoi: Message Format for Interactive Mode”

-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 to modify the default message format in interactive mode. The following formats are supported (example source file x:\Freescale\mysourcefile.cpph):

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%s</td>
<td>Source Extract</td>
<td>x:\Freescale\</td>
</tr>
<tr>
<td>%p</td>
<td>Path</td>
<td></td>
</tr>
</tbody>
</table>
Burner Options

%f  Path and name          x:\Freescale\mysourcefile
%n  File name                mysourcefile
%e  Extension               .cpph
%N  File (8 chars)          mysource
%E  Extension (3 chars)    .cpp
%l  Line                     3
%c  Column                   47
%o  Pos                       1234
%K  Uppercase kind          ERROR
%k  Lowercase kind          error
%d  Number                   B1815
%m  Message                  text
%  Percent                  %
\n New line

Example:

-WmsgFoi"%f%e(%l): %k %d: %m\n"

produces a message in following format:

X:\C.C(3): information B2901: Message

See also:

- "ERRORFILE: Error File Name Specification"
- "-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode"
- "-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode"
- "-WmsgFonp: Message Format for No Position Information"
- "-WmsgFob: Message Format for Batch Mode"

-WmsgFonf: Message Format for No File Information

Group:

MESSAGE

Syntax:

"-WmsgFonf"<string>.

Arguments:

<string>: format string (see below).
Burner Options

Default:

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

Description:

Sometimes there is no file information available for a message (e.g. if a message is not related to a specific file). Then this message format string is used. The following formats are supported:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%K</td>
<td>Uppercase kind</td>
<td>ERROR</td>
</tr>
<tr>
<td>%k</td>
<td>Lowercase kind</td>
<td>error</td>
</tr>
<tr>
<td>%d</td>
<td>Number</td>
<td>B1815</td>
</tr>
<tr>
<td>%m</td>
<td>Message</td>
<td>text</td>
</tr>
<tr>
<td>%%</td>
<td>Percent</td>
<td>%</td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
<td></td>
</tr>
</tbody>
</table>

Example:

-WmsgFonf"%k %d: %m\n"
produces a message in following format:
information B1034: Message

See also:

- "ERRORFILE: Error File Name Specification"
- "-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode"
- "-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode"
- "-WmsgFonp: Message Format for No Position Information"
-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 is not related to a certain position). Then this message format string is used. The following formats are supported:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%K</td>
<td>Uppercase kind</td>
<td>ERROR</td>
</tr>
<tr>
<td>%k</td>
<td>Lowercase kind</td>
<td>error</td>
</tr>
<tr>
<td>%d</td>
<td>Number</td>
<td>B1815</td>
</tr>
<tr>
<td>%m</td>
<td>Message</td>
<td>text</td>
</tr>
<tr>
<td>%</td>
<td>Percent</td>
<td>%</td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
<td></td>
</tr>
</tbody>
</table>

Example:
-WmsgFonp"%k %d: %m\n"
produces a message in following format:
information B1324: Message
Burner Options

See also:

- “ERRORFILE: Error File Name Specification”
- “-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode”
- “-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode”
- “-WmsgFonp: Message Format for No Position Information”

-WmsgNe: Number of Error Messages

Group:
MESSAGE

Syntax:

“-WmsgNe“ <number>.

Arguments:

<number>: Maximum number of error messages.

Default:

50

Description:
Use this option to set the number of errors allowed until the application stops processing.

Example:

-WmsgNe2

The application stops after two error messages.

See also:

- “-WmsgNi: Number of Information Messages”
- “-WmsgNw: Number of Warning Messages”
**-WmsgNi: Number of Information Messages**

**Group:**
MESSAGE

**Syntax:**
```
-WmsgNi "<number>
```

**Arguments:**

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

**Default:**
50

**Description:**
Use this option to set the number of information messages.

**Example:**

-WmsgNi 10

Only ten information messages are logged.

**See also:**
- `-WmsgNe: Number of Error Messages`
- `-WmsgNw: Number of Warning Messages`

---

**-WmsgNu: Disable User Messages**

**Group:**
MESSAGE

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

---
Burner Options

Arguments:

“a” : Disable messages that relate to include files
“b” : Disable messages that relate to files being read (input files)
“c” : Disable messages that relate to generated files
“d” : Disable messages that relate to processing statistics (code size, RAM/ROM usage, etc.)
“e” : Disable informal messages

Default:
None.

Description:
Some messages produced by the application are not in the normal message categories (WARNING, INFORMATION, ERROR, FATAL). With this option such messages can be disabled. This option reduces the number of messages and simplifies error parsing of other tools.

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

Example:
-WmsgNu=c

-WmsgNw: Number of Warning Messages

Group:
MESSAGE

Syntax:

“-WmsgNw” <number>.

Arguments:

<number> : Maximum number of warning messages.

Default:
50
**Burner Options**

---

**Description:**
Sets the number of warning messages.

**Example:**
```
-WmsgNw15
```
Only 15 warning messages are logged.

**See also:**
- 
  - `-WmsgNe: Number of Error Messages`
  - `-WmsgNi: Number of Information Messages`

---

**-WmsgSd: Setting a Message to Disable**

**Group:**
MESSAGE

**Syntax:**
```
-WmsgSd <number>.
```

**Arguments:**

<number>: Message number to be disabled, for example: 1801

**Default:**
None.

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

**Example:**
```
-WmsgSd1801
```
disables the message for implicit parameter declaration.

**See also:**
- 
  - `-WmsgSi: Setting a Message to Information`
  - `-WmsgSw: Setting a Message to Warning`
  - `-WmsgSe: Setting a Message to Error`
Burner Options

-WmsgSe: Setting a Message to Error

Group:
MESSAGE

Syntax:
"-WmsgSe" <number>.

Arguments:
<number>: Message number to be set as an error message, e.g. 1853

Default:
None.

Description:
Changes message category for a message to error message category.

Example:
-WmsgSe1853

See also:
- "-WmsgSd: Setting a Message to Disable"
- "-WmsgSi: Setting a Message to Information"
- "-WmsgSw: Setting a Message to Warning"

-WmsgSi: Setting a Message to Information

Group:
MESSAGE

Syntax:
"-WmsgSi" <number>.

-
Arguments:

<number>: Message number to be set as an information message, e.g. 1853

Default:
None.

Description:
Changes the category for a message to information message category.

Example:
-WmsgSi1853

See also:
- `-WmsgSd: Setting a Message to Disable`
- `-WmsgSw: Setting a Message to Warning`
- `-WmsgSe: Setting a Message to Error`

-WmsgSw: Setting a Message to Warning

Group:
MESSAGE

Syntax:

-WmsgSw  <number>.

Arguments:

<number>: Message number to be set as a warning, e.g. 2901

Default:
None.

Description:
Sets message category for a message to warning message.

Example:
-WmsgSw2901
Burner Options

See also:
- “-WmsgSd: Setting a Message to Disable”
- “-WmsgSi: Setting a Message to Information”
- “-WmsgSe: Setting a Message to Error”

-WOutFile: Create Error Listing File

Group:
MESSAGE

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

Arguments:
None.

Default:
Error list file is created.

Description:
This option controls the creation of an error log file. The file contains a list of all messages and errors encountered during processing. Since text error feedback can also be handled with pipes to the calling application, it is possible to obtain this feedback without an explicit file. The name of the list file is controlled by the environment variable ERRORFILE.

Example:
-WOutFileOn
The error file specified by ERRORFILE is created.
-WOutFileOff
No error file is created.

See also:
- “-WErrFile: Create “err.log” Error File”
- “-WStdout: Write to Standard Output”
-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 text written to an error file can also be written to stdout.

Example:

-WStdoutOn
All messages are written to stdout.
-WErrFileOff
Nothing is written to stdout.

See also:

• "-WErrFile: Create “err.log” Error File”
• "-WOutFile: Create Error Listing File”
Burner Options

-W1: No Information Messages

Group:
MESSAGE

Syntax:
"-W1".

Arguments:
None.

Default:
None.

Description:
This option excludes INFORMATION messages, only WARNING and ERROR messages are generated.

Example:
-W1

See also:
- "-WmsgNi: Number of Information Messages"

-W2: No Information and Warning Messages

Group:
MESSAGE

Syntax:
"-W2".

Arguments:
None.
Default:
None.

Description:
This option suppresses all INFORMATION and WARNING messages, only ERRORs are generated.

Example:
-w2

See also:
- ^-WmsgNi: Number of Information Messages^”
- ^-WmsgNw: Number of Warning Messages^”
Burner Options
Environment Variables

This chapter describes the environment variables used by the application. Some environment variables are also used by other tools (e.g. Linker).

Various parameters may be set in an environment using environment variables. The syntax is:

Parameter = KeyName "=" ParamDef.

**NOTE** Normally 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

Parameters may be defined in several ways:

- Using system environment variables supported by your operating system.
  
  Put definitions in a file called `DEFAULT.ENV` (`.hidefaults` for UNIX) in the default directory.

  **NOTE** The maximum length of environment variable entries in the `DEFAULT.ENV` or `.hidefaults` is 65535 characters.

- Put definitions in a file specified by the system environment variable `ENVIRONMENT`.

  **NOTE** The default directory mentioned above can be set by the system environment variable `"DEFAULTDIR: Default Current Directory"`.

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: Set Environment Variable" option. Ensure that no spaces are at the end of environment variables.
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 (e.g. DEFAULT.ENV / or .hidefaults).

Normally, the current directory of an executed tool is determined by the operating system or program that launches another one, for example WinEdit.

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

For MS Windows 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 one 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 current directory specification (e.g. WinEdit), the current directory is the one specified by the launching tool (e.g. current directory definition in WinEdit).
- The current directory is where the local project file is located. Changing the current project file also changes the current directory, if the other project file is in a different directory. Note that browsing for a C file does not change the current directory.

To overwrite this behavior, the environment variable “DEFAULTDIR: Default Current Directory” may be used.

The current directory is displayed along with other information in the about box or with the option “-v”.
Global Initialization File (MCUTOOLS.INI) (PC only)

All tools may store global data in MCUTOOLS.INI. A tool 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 (e.g. C:\WINDOWS).

Example:
- C:\WINDOWS\MCUTOOLS.INI
- D:\INSTALL\PROG\MCUTOOLS.INI

If a tool is started in D:\INSTALL\PROG directory, the project file in the same directory as the tool is used (D:\INSTALL\PROG\MCUTOOLS.INI).

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

The following section provides a short description of entries in the MCUTOOLS.INI file:

[Installation] Section

Path

Arguments:
Last installation path.

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

Example:
Path=c:\install
Environment Variables

---

Group

**Arguments:**
Last installation program group.

**Description:**
When a tool is installed, this variable stores the installation program group created.

**Example:**
Group=Burner

[Options] Section

---

DefaultDir

**Arguments:**
Default Directory to be used.

**Description:**
Specifies the current directory for all tools on a global level (see also, environment variable "DEFAULTDIR: Default Current Directory").

**Example:**
DefaultDir=c:\install\project
**Environment Variables**

---

**[Burner] Section**

**SaveOnExit**

**Arguments:**

1/0

**Description:**

1 if the configuration should be stored when the tool is closed, 0 if it should not be stored. The tool does not ask to store a configuration, in either case.

---

**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.

---

**SaveEditor**

**Arguments:**

1/0

**Description:**

1 if the visible topics should be stored when writing a project file, 0 if not. The editor setting contains all information from the editor configuration dialog.
Environment Variables

SaveOptions

**Arguments:**

1/0

**Description:**

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

RecentProject0, RecentProject1...

**Arguments:**

Names of 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.

**Example:**

```
SaveOnExit=1
SaveAppearance=1
SaveEditor=1
SaveOptions=1
RecentProject0=C:\myprj\project.ini
RecentProject1=C:\otherprj\project.ini
```
[Editor] Section

Editor_Name

**Arguments:**

The name of the global editor

**Description:**

Specifies the name displayed for the global editor. This entry provides only a description. Its content is not used to start the editor.

This entry cannot be modified with the tool.

Editor_Exe

**Arguments:**

Name of executable file for global editor

**Description:**

Specifies file called when the global editor setting is active. In the editor configuration dialog, the global editor selection is active when this entry is present and not empty.

Editor_Opts

**Arguments:**

Options to use the global editor

**Description:**

Specifies options that should be used with the global editor. If this entry is not present or empty, "%f" is used. The command line to launch the editor is built by taking the Editor_Exe content, then appending a space followed by this entry.
Environment Variables

Example:

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

Example

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

[Installation]
Path=c:\Freescale
Group=Burner

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

[Options]
DefaultDir=c:\myprj

[Burner]
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 tools does not change the default.env file. Its content is only read. All configuration properties are stored in the configuration file. The same configuration file can 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 tool. Only when the shell uses the same file as the tool, the editor configuration written and maintained by the shell can be used by the tool. Apart from this, the tool can use any file name for the project file.
Environment Variables

The configuration file has the same format as windows .ini files. The application stores its own entries with the same section name as in the global mcutools.ini file. The application backend is encoded into the section name, so that different application backends can use the same file without overlapping. Different versions of the same tool use the same entries. This 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 different tool versions. If no incompatible options are enabled when the file is last saved, the same file may be used for both tool versions.

The current directory is always the directory containing the configuration. If a configuration file in a different directory is loaded, then the current directory also changes. When the current directory changes, the default.env file is reloaded.

At startup there are two ways to load a configuration:

- use the command line option 

- from the project.ini file in the current directory

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

[Editor] Section

Editor_Name

Arguments:

Name of the local editor

Description:

Specifies the name displayed for the local editor. Its content is not used to start the editor.

Saved:

This entry has the same format as global editor configuration in the mcutools.ini file.
Environment Variables

Editor_Exe

**Arguments:**
Name of executable file for local editor

**Description:**
Specifies 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:**
This entry has the same format as the global editor configuration in the mcutools.ini file.

Editor_Opts

**Arguments:**
Options to use with the local editor

**Description:**
Specifies options to use with local editor. If this entry is not present or empty, "%f" is used. The command line to launch the editor is built by taking the Editor_Exe content, then appending a space followed by this entry.

**Saved:**
This entry has the same format as the global editor configuration in the mcutools.ini file.

**Example:**

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

[Burner] Section

RecentCommandLineX, X= integer

Arguments:
String with a command line history entry, e.g. "fibo.bbl"

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

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

CurrentCommandLine

Arguments:
String with the command line, e.g. "-ffibo.bbl -w1"

Description:
The currently visible command line content.

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

StatusBarEnabled

Arguments:
1/0

Special:
This entry is only considered at startup.
Environment Variables

**Description:**
1: the statusbar is visible
0: the statusbar is hidden

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

---

**ToolbarEnabled**

**Arguments:**
1/0

**Special:**
This entry is only considered at startup.

**Description:**
1: the toolbar is visible
0: the toolbar is hidden

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

---

**WindowPos**

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

**Special:**
This entry is only considered at startup.
Entry changes do not show “*” in the title.

**Description:**
These numbers contain the position and state of the window (maximized, etc.) and other flags.
**Environment Variables**

---

**Saved:**

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

---

### 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
```
Environment Variables

ShowTipOfDay

Arguments:
0/1

Description:
Display Tip of the Day dialog at startup.
1: show
0: hide (can be displayed from the help menu)

Saved:
Always when saving a configuration file.

Options

Arguments:
-W2

Description:
The currently active option string. This entry can be long, since messages are also contained here.

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

EditorType

Arguments:
0/1/2/3

Description:
This entry specifies which editor configuration is active.
Environment Variables

0: global editor configuration (in mcutools.ini file)
1: local editor configuration
2: command line editor configuration, entry EditorCommandLine
3: DDE editor configuration, entries beginning with EditorDDE

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

EditorCommandLine

Arguments:
Command line, for WinEdit: "C:\Winapps\WinEdit.exe %f /#::%l"

Description:
Command line to open a file.

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

EditorDDEClientName

Arguments:
Client commend, e.g. "\open(%f)"

Description:
Name of client for DDE editor configuration.

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

EditorDDETopicName

**Arguments:**
Topic name, e.g. "system"

**Description:**
Name of topic for DDE editor configuration.

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

EditorDDEServiceName

**Arguments:**
Service name, e.g. "system"

**Description:**
Name of service for DDE editor configuration.

**Saved:**
Only with Editor Configuration set in the File->Configuration Save Configuration dialog.
Burner Dialog Entries in [BURNER]

BurnerUndefByte

**Arguments:**
Integral value of undefined bytes. Default is 0xff.

**Description:**
Value of the Undef Byte entry on the Content page in the Burner dialog.

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

BurnerSwapByte

**Arguments:**

- 0: do not swap
- 1: swap

**Description:**
Value of the Swap Bytes check box on the Content page in the Burner dialog.

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

BurnerOrigin

Arguments:
Integral value (0,1,2...)

Description:
Value of the Origin field on the Content page in the Burner dialog.

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

BurnerDestination

Arguments:
Integral value (0,1,2...)

Description:
Value of the Destination Offset field on the Content page in the Burner dialog.

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

BurnerLength

Arguments:
Integral value (0,1,2...)

Description:
Value of the Length field on the Content page in the Burner dialog.

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

Arguments:
0: Freescale S
1: Intel Hex
2: Binary

Description:
Format type specified on the Content page in the Burner dialog.

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

BurnerDataBus

Arguments:
0: “1 Byte”
1: “2 Bytes”
2: “4 Bytes”
Not the size in bytes.

Description:
Setting in the Data Bus field on the Content page in the Burner dialog.

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

---

**BurnerOutputType**

**Arguments:**
0: “Com1”
1: “Com2”
2: “Com3”
3: “Com4”
4: “File”

**Description:**
Setting in the Output field on the Input/Output page in the Burner dialog.

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

---

**BurnerDataBits**

**Arguments:**
0: “7 Bits”
1: “8 Bits”

**Description:**
Setting in the Data Bits field on the Input/Output page in the Burner dialog.

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

Arguments:
0: “None”
1: “Odd”
2: “Even”

Description:
Setting in the Parity field on the Input/Output page in the Burner dialog.

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

BurnerByteCommands

Arguments:
0: “1st Byte (msb)”
1: “2nd Byte”
2: “3rd Byte”
3: “4th Byte”
4: “1st Word”
5: “2nd Word”

Description:
Setting in the command box on the Input/Output page in the Burner dialog.

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

BurnerBaudRate

**Arguments:**
300, 600, 1200, 2400, 4800, 9600, 19200, 38400

**Description:**
Setting in the Baud Rate box on the Input/Output page in the Burner dialog.

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

BurnerOutputFile

**Arguments:**
File Name. E.g. “file.s19”.

**Description:**
Content of the Name box on the Input/Output page in the Burner dialog.

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

BurnerHeaderFile

**Arguments:**
File Name. E.g. “headerfile”.

**Description:**

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

Arguments:

File Name. E.g. “file.abs”.

Description:


Saved:

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

Configuration File 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

[Burner]
StatusBarEnabled=1
ToolbarEnabled=1
WindowPos=0,1,-1,-1,-1,390,107,1103,643
WindowFont=-16,500,0,Courier
TipFilePos=0
ShowTipOfDay=1
Options=-w1
EditorType=3
RecentCommandLine0=-ffibo.bbl -w1
CurrentCommandLine=-ffibo.bbl -w2
EditorDDECLientName=[open(%f)]
EditorDDETopCName=system
EditorDDEServiceName=msdev
EditorCommandLine=C:\WinEdit\WinEdit.exe %f /#:%l
BurnerUndefByte=255
BurnerSwapByte=0
BurnerOrigin=0
BurnerDestination=0
BurnerLength=65536
BurnerFormat=0
BurnerDataBus=0
```
Environment Variables

BurnerOutputType=4
BurnerDataBits=1
BurnerParity=0
BurnerByteCommands=0
BurnerBaudRate=9600
BurnerOutputFile=outputfile.s19
BurnerHeaderFile=headerfile
BurnerInputFile=InputFile.abs

Paths

Most environment variables contain path lists indicating where to look for files. A path list is a list of directory names separated by semicolons.

PathList = DirSpec {";" DirSpec}.
DirSpec = ["*" ] DirectoryName.

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

- If a directory name is preceded by an asterisk ("*"), the programs recursively search the directory tree for a file, not just the given directory. 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.

We recommend working with WinEdit and setting the environment by means of a DEFAULT.ENV (.hidefaults for UNIX) file in your project directory. This 'project directory' can be set in WinEdit's 'Project Configure...' menu command. This way, you can have different projects in different directories, each with its own environment.

NOTE When using WinEdit, do not set the system environment variable "DEFAULTDIR: Default Current Directory". If you do and this variable does not contain the project directory given in WinEdit’s project configuration, files might not be put where you expect them.
Line Continuation

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

**Line Continuation Example:**

```bash
OPTIONS=\ 
-W2 \ 
-Wpd
```

This is the same as

```bash
OPTIONS=-W2 -Wpd
```

Be careful using this feature with paths, e.g.

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

will result in

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

To avoid such problems, we recommend using a semicolon ‘;’ at the end of a path, if there is a ‘\’ at the end:

```bash
GENPATH= . \; 
TEXTFILE= . \txt
```
Environment Variables

Environment Variable Details

The remainder of this section describes each of the environment variables available for the burner. Variables are listed in alphabetical order and each is divided into several sections.

Table 18.1 Environment Variable Detail Parameters

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used By</td>
<td>Lists tools that use variable</td>
</tr>
<tr>
<td>Synonym</td>
<td>For some environment variables a synonym also exists. The synonyms may be used for older releases of the tool and will be removed in the future. A synonym has lower precedence than the environment variable.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Specifies syntax of option in EBNF format.</td>
</tr>
<tr>
<td>Arguments</td>
<td>Describes and lists optional and required arguments for the variable.</td>
</tr>
<tr>
<td>Default</td>
<td>Shows default setting for the variable or none.</td>
</tr>
<tr>
<td>Description</td>
<td>Provides a detailed description of the option and how to use it.</td>
</tr>
<tr>
<td>Example</td>
<td>Gives an example of usage, and effects of the variable where possible. Examples show an entry in default.env for PC or in the .hidefaults file for UNIX.</td>
</tr>
<tr>
<td>See also</td>
<td>Related sections.</td>
</tr>
</tbody>
</table>
**DEFAULTDIR: Default Current Directory**

**Used By:**
Compiler, Assembler, Linker, Decoder, Debugger, Libmaker, Maker, Burner

**Synonym:**
None.

**Syntax:**
```
"DEFAULTDIR=" <directory>.
```

**Arguments:**

<directory>: The default current directory.

**Default:**
None.

**Description:**
This environment variable specifies the default directory for all tools. All tools indicated above will use the directory specified as their current directory instead of the one defined by the operating system or launching tool (e.g. editor).

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

**Example:**
```
DEFAULTDIR=C:\INSTALL\PROJECT
```
Environment Variables

ENVIRONMENT: Environment File Specification

Used By:
Compiler, Linker, Decoder, Debugger, Libmaker, Maker, Burner

Synonym:
HIENVIRONMENT

Syntax:
"ENVIRONMENT=" <file>.

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

Default:
None.

Description:
This variable is specified at the system level. Normally the application looks in the The Current Directory for an 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 a system level (global) environment variable. It CANNOT be specified in a default environment file (DEFAULT.ENV/.hidefaults).

Example:
ENVIRONMENT=\Freescale\prog\global.env
ERRORFILE: Error File Name Specification

Used By:
Compiler, Assembler, Linker, Burner

Synonym:
None.

Syntax:
“ERRORFILE=” <file name>.

Arguments:
<file name>: File name with possible format specifiers.

Description:
This environment variable specifies the name of the error file. Possible format specifiers are:
'%n': Substitutes with file name without path.
'%p': Path of the source file.
'%f': Full file name, including path (same as '%p%n')

In case of an invalid error file name, a notification box is shown.
Environment Variables

Example:

ERRORFILE=MyErrors.err
lists all errors into the file MyErrors.err in current 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, and placed in the same directory as the source file. For example, if we process a file \sources\test.c, an error list file \sources\test.err will be generated.

ERRORFILE=\dir1\%n.err
For a source file test.c, an error list file \dir1\test.err will be generated.

ERRORFILE=%p\errors.txt
For source file \dir1\dir2\test.c, an error file \dir1\dir2\errors.txt will be generated.

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

Example:

This example shows usage of this variable to support correct error feedback with the WinEdit Editor, which looks for an error file called EDOUT:

Installation directory: E:\INSTALL\PROG
Project sources: D:\MEPHISTO
Common Sources for projects: E:\CLIB

Entry in default.env (D:\MEPHISTO\DEFAULT.ENV):
ERRORFILE=E:\INSTALL\PROG\EDOUT

Entry in WINEDIT.INI (in Windows directory):
OUTPUT=E:\INSTALL\PROG\EDOUT
**GENPATH: #include “File” Path**

**Used By:**
Compiler, Linker, Decoder, Debugger, Burner

**Synonym:**
HIPATH

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

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

**Default:**
Current directory

**Description:**
This path specification is used by the burner to search for input files.

**NOTE**
If a directory specification in this environment variable starts with an asterisk ("*"), the complete directory tree is searched recursively. All subdirectories are searched. Within one level in the directory tree, search order of the subdirectories is random undeterminable.

**Example:**
GENPATH=\sources\include;..\..\headers;\usr\local\lib
Environment Variables

TMP: Temporary Directory

Used By:
Compiler, Assembler, Linker, Debugger, Libmaker, Burner

Synonym:
None.

Syntax:
"TMP=" <directory>.

Arguments:
<directory>: Directory 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 a system level (global) environment variable. It CANNOT be specified in a default environment file (DEFAULT.ENV / .hidefaults).

Example:
TMP=C:\TEMP
Burner Messages

This chapter describes messages produced by the Application. Because of the number of messages produced, some may not have been documented at the time of this release.

Message Kinds

There are five kinds of messages generated:

**Information**
A message will be printed and compilation will continue. Information messages are used to indicate actions taken by the application.

**WARNING**
A message will be printed and processing will continue. Warning messages are used to indicate possible programming errors.

**ERROR**
A message will be printed and processing is stopped. Error messages are used to indicate illegal use of the language.

**FATAL**
A message will be printed and processing is aborted. A fatal message indicates a severe error that will stop processing.

**DISABLE**
No message will be issued and processing will continue. The application ignores this type of message.
Message Details

If the application prints a message, the message contains a message code and a four to five digit number. This number may be used to search for the indicated message. Following message codes are supported:

- “A” for Assemblers
- “B” for Burner
- “C” for Compilers
- “L” for Linker
- “LM” for Libmaker
- “M” for Maker

All messages generated by the application are documented in increasing number order for quick retrieval.

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 message is also noted, e.g. [ERROR] indicates that the message is an error message.

[DISABLE, INFORMATION, WARNING, ERROR] indicates that the message is a warning message by default, but the user might change the message to either DISABLE, INFORMATION or ERROR.

After the message type, there may be an additional entry indicating the related language:

- C++: Message is generated for C++
- M2: Message is generated for Modula-2
Message List

The following pages describe all messages.

B1: Unknown Message Occurred

[FATAL]

Description:
The application tried to emit a message which was not defined. This is an internal error that should not occur. Please report any occurrences to your distributor.

B2: Message Overflow, Skipping <kind> Messages

[DISABLE, INFORMATION, WARNING, ERROR]

Description:
Application displayed the number of messages as controlled by the burner options:

- `-WmsgNi: Number of Information Messages`
- `-WmsgNw: Number of Warning Messages`
- `-WmsgNe: Number of Error Messages`

Further options of this kind are not displayed.

TIP Use the options `-WmsgNi`, `-WmsgNw` and `-WmsgNe` to change the number of messages of whatever type the utility will accept.

B50: Input file ‘<file>’ not found

[FATAL]

Description
The Application was not able to find a file needed for processing.
Burner Messages
Message List

TIP  Check if the file really exits. Check if you are using a file name containing spaces (in this case you have to place quotes around filename).

B51: 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 support statistic log files. Even if a tool does not support it, the message still exists, but never issued.

B52: Error in Command Line '<cmd>

[FATAL]

Description
In case there is an error while processing the command line, this message is issued.

B64: Line Continuation Occurred in <FileName>

[DISABLE, INFORMATION, WARNING, ERROR]

Description:
In an environment file, the character '\ at the end of a line is interpreted as line continuation. This line and the next one are interpreted as one line. Because the path separation character of MS-DOS is also '\, paths are often incorrectly written that end with '\. Instead use a '.' after the last '\ in a path.
Example:

Current Default.env:

... LIBPATH=c:\Freescale\lib\ OBJPATH=c:\Freescale\work ...

Is interpreted as ...

... LIBPATH=c:\Freescale\libOBJPATH=c:\Freescale\work ...

TIP To fix it, append a \ at the end of path:

... LIBPATH=c:\Freescale\lib\. OBJPATH=c:\Freescale\work\ ...

NOTE Because this information occurs during the initialization phase of the application, the message prefix might not occur in the error message. So it might occur as "64: Line Continuation occurred in <FileName>".

B65: Environment Macro Expansion Error '<description>' for <variablename>

[DISABLE, INFORMATION, WARNING, ERROR]

Description:
During an environment variable macro substitution a problem occurred. Possible causes could be that the named macro did not exist or some length limitation was reached. Recursive macros may also cause this message.
Example:

Current variables:

...  
LIBPATH=${LIBPATH}  
...  

TIP  Check the definition of the environment variable.

---

B66: Search Path <Name> Does Not Exist

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The tool searched for a file that was not found. During the failed search, a non existing path was encountered.

TIP

- Check the spelling of your paths.
- Update the paths when moving a project.
- Use relative paths in your environment variables.
- Check if network drives are available.

---

B1000: Could Not Open '<FileType>' '<File>

[ERROR]

Description:

The specified file could not be open.
This message is used for input and output file.

TIP  For files to be generated, check if they are modifiable and whether sufficient space exists on the disk. Ensure that the file is not locked by another application and that the path exists.
B1001: Error in Input File Format

[ERROR]

Description:
An error occurred while reading the input file.

TIP
- Try to generate the input file again.
- Check if you have enough free disk space.

B1002: Selected Communication Port is Busy

[ERROR]

Description:
The selected communication port can not be accessed.

TIP
1. Check if another application has locked the serial port.
2. Check if the correct serial port is specified.

B1003: Timeout or Failure for the Selected Communication

[ERROR]

Description:
There was a timeout or general failure on the selected communication port.

TIP
Check if another application has locked the serial port.
B1004: Error in Macro ‘<macro>’ at Position <pos>: ‘<msg>’

[ERROR]

**Description:**
While resolving a macro, the Burner was not able to resolve it. A macro is surrounded by ‘%’ characters (e.g. %ABS_FILE%).

**TIP**
Check if you have a definition of your macro in the environment. Check if the macro is passed on the command line using the -Env option.

B1005: Error in Command Line at Position <pos>: ‘<msg>’

[ERROR]

**Description:**
If the command line scanner detects an invalid command line, this message is produced.

**TIP**
Check the syntax of your command line.

B1006: ‘<msg>’

[ERROR]

**Description:**
This message is used to indicate generic errors.
Libmaker

Introduction

This document describes the Libmaker, a utility program for creating and maintaining object file libraries. Using libraries can speed up linking since fewer files are involved, and also helps in structuring large applications.

Libraries may be given in the linker parameter file instead of object files.

This section consists of the following chapters:

- “Libmaker Interface”: Description of the GUI.
- “Environment Variables”: Environment variables used by the tool.
- “Libmaker Options”: Description with examples of option settings for the tool.
- “Libmaker Messages”: Description with examples of messages issued by the tool.
- “EBNF Notation”: Overview of the Extended Backus–Naur Form notation.

Product Highlights

- User Interface
- On-line Help
- Flexible Message Management
- 32-bit Application
- Builds libraries with Freescale (former Hiware) or ELF/DWARF object files
Starting the Libmaker Utility

You can start tools (compiler/linker/assembler/decoder/...) using:

- Windows Explorer
- Icon on the desktop
- Icon in a program group
- Batch/command files
- Other tools (Editor, Visual Studio)

All of the utilities described in this book may be started from executable files located in
the Prog folder of your IDE installation. The executable files are:

- maker.exe  Maker: The Make Tool
- burner.exe  The Burner Utility
- decoder.exe The Decoder
- libmaker.exe Libmaker

With a standard full installation of the HC(S)08/RS08 CodeWarrior IDE, the executable
files are located at:

```
C:\Program Files\Freescale\CW08 v5.x\Prog
```

To start the Libmaker Utility, you can click on libmaker.exe. The Libmaker Default
Configuration window appears:

User Interface

Libmaker provides:

- Graphical User Interface
- Command-Line User Interface
- Online Help
- Error Feedback
- Easy integration with other tools (e.g. CodeWarrior IDE, CodeWright, MS Visual
  Studio, WinEdit)
Interactive Mode

If you start the libmaker with no input (no options or input files), then the graphical user interface is active (interactive mode). This is usually the case if you start the tool using Explorer or an icon.
Libmaker Interface

Startup Command Line Options

There are special options for tools which can only be specified at tool startup (while launching the tool), e.g. they cannot be specified interactively:

- `-Prod: Specify Project File at Startup` can be used to specify the current project directory or file, for example `libmaker.exe -Prod=c:\Freescale\demo\myproject.pjt`

There are other options used to launch the tool and open its dialog boxes. Those dialogs are available in the compiler/assembler/burner/maker/linker/decoder/libmaker:

- `-ShowOptionDialog`: This startup option opens the tool option dialog.
- `-ShowMessageDialog`: This startup option opens the tool message dialog.
- `-ShowConfigurationDialog`: This opens the `File->Configuration` dialog.
- `-ShowBurnerDialog`: Opens burner dialog (burner only)
- `-ShowSmartSliderDialog`: Opens smart slider dialog (compiler only)
- `-ShowAboutDialog`: Opens the tool about box.

The above mentioned options will open a dialog where you can specify tool settings. If the OK button is pressed in the dialog, settings are stored in the current project settings file. Example usage:

c:\\Freescale\\prog\\libmaker.exe -ShowOptionDialog -Prod=c:\demos\myproject.pjt

Command Line Interface

Libmaker provides both a command line interface and an interactive interface. If no arguments are given on the command line, a window appears.

Libmaker Commands

When Libmaker is started, it opens a window and prompts for arguments. The arguments may be given on a command line in the following format:

```
LibCommand = Creation
```

LibCommand

- `Creation`
Libmaker Interface
Command Line Interface

| AppendFiles
| RemoveFiles
| List
| "@" FileName.

Creation = FileName AddList "=" LibName.
AddList = ("+" FileName).
AppendFile = LibName AddList "=" LibName.
RemoveFiles = LibName SubList ["=" LibName].
SubList = "-" FileName {"-" FileName}.
List = LibName "?" FileName.

Libmaker uses the environment variable OBJPATH when looking for object or library files, or writing the library file. The environment variable TEXTPATH is used when looking for a command file or writing the listing file.

Managing Libraries

All the commands below are supposed to be in a libmaker command file (text file with the commands in it, line by line). Alternatively you can pack the commands into the "-Cmd: Libmaker Commands" option and pass it to the libmaker directly (e.g. from a make file).
For example:

a.o + b.o = c.lib

can be written as an option to the libmaker as:
libmaker.exe -Cmd(a.o + b.o = c.lib)

As it is difficult to create a command line with the ‘+’ operator in a make utility, the libmaker supports as well the alternative syntax without the ‘+’ operator:

-Cmd(a.o + b.o = c.lib)

can be written as well as:

-Cmd(a.o b.o = c.lib)

Building a Library

Building a library collects all the given object files and/or libraries into one new library given after the "=" sign:

file1.o + file2.o + mylib.LIB = ourlib

NOTE    To create a library, there must be at least two files left of the equal sign.
**Adding Files to a Library**

Adding files to an existing library works the same as building a library:

```
ourlib.LIB + file3.o = ourlib
```

**Removing a File from a Library**

It is also possible to remove one or more files from a library:

```
ourlib.LIB - file1.o = ourlib
```

This removes the object file `file1.o` from the library.

**Creating a New Library**

It is also possible to create a new library:

```
ourlib - file1.o = hislib
```

In this case, the original library `ourlib` is *not* overwritten.

**Extracting a File from a Library**

The code line:

```
LibName * ObjName
```

Extracts the object file named `ObjName` from the library. No path is given with the argument `ObjName`. The object file is written to the same directory as the library. The file is not removed from the library. An existing object file with the same name as an extracted object file is overwritten without warning.

Example:

```
mylib.lib * myobj.obj
```

writes the object file `myobj.obj` into the same directory as `mylib.lib`.

**Listing the Contents of a Library**

Libmaker also generates an alphabetically sorted list of all exported objects in the library. Enter name of library:

```
ourlib.LIB
```

The list file has the same name as the library, but with extension `.LST`. If you want to specify a different name, enter:

```
ourlib.LIB ? mylist.TXT
```
Command Files

Libmaker also supports command files. A command file is a text file containing commands for the libmaker. To use a command file, enter:

@mycmds.CMD

The libmaker reads the file and interprets the commands line by line.

Batch Mode

If you start the tool with arguments (options and/or input files), then the tool is started in batch mode. For example, you can specify the following line:

C:\Freescale\PROG\libmaker.exe @mycommands.txt

In batch mode, the tool does not open a window. It is displayed in the taskbar during the time the input is processed and terminates afterwards.

Because it is possible to start 32-bit applications from the command line (e.g. DOS prompt under Windows NT/95/98), you can simply type the commands you want to execute:

C:\> C:\Freescale\PROG\libmaker.exe -cmd(a.o b.o = c.lib)

You can redirect the message output (stdout) of a tool using the normal redirection operators, e.g. ‘>’ to write the message output to a file:

C:\> C:\Freescale\PROG\libmaker.exe -h > myoutput.txt

You will notice that the command line process immediately returns after starting the tool process. It does not wait until the started process has finished. To start a process and wait for termination (e.g. for synchronization) you can use the ‘start’ command under Windows NT/95/98 and the ‘/wait’ option (see windows help: ‘help start’ for more information):

C:\> start /wait C:\Freescale\PROG\libmaker.exe -cmd(a.o b.o = c.lib)

Using ‘start /wait’ you can write batch files to process your files.

If you need to redirect the libmaker output to stderr/stdout on your DOS shell, then you need the piper utility:

C:\> C:\Freescale\PROG\piper.exe
C:\Freescale\PROG\libmaker.exe -h

This will direct all the messages to the DOS shell.
Libmaker Graphic User Interface

The Libmaker Default Configuration window appears when you do not specify a file name while starting the application. This window contains a menu bar, tool bar, content area, and status bar.

Figure 20.1 Libmaker Default Configuration Window
Libmaker Interface
Libmaker Graphic User Interface

Libmaker Default Configuration Window
The Libmaker Default Configuration window title displays the application name and project name. If no project is loaded, “Default Configuration” is displayed. A “*” after the configuration name indicates that some values have changed.

NOTE Not only option changes, but editor configuration and appearance can cause the “*” to appear.

Window Content Area
The content area is used as a text container, where logging information about the process session is displayed. This information consists of:

- name of file being processed
- name (including full path) of files processed (main C file and all files included)
- list of error, warning and information messages generated
- size of code generated during the process session

When a file is dropped into the application window content area, the corresponding file is either loaded as configuration or processed. It is loaded as configuration if the file has the extension “ini”. If not, the file is processed with the current option settings.

Text in the application window content area will display the following information.

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

File information is available for text file output. Information is available for all source and include files, and messages. If file information is available, double-clicking on the text or message opens the file in an editor; as specified in the editor configuration. Also, a context menu can be opened with the right mouse button. The menu contains an “Open...” entry if file information is available. If a file can not be opened although a context menu entry is present, see the “Configuration Window Editor Settings Tab” section.

The message number is available for any message output. 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 does not have a message number, main help is displayed.
- Right-click on the message and select “Help on...”. This entry is only available if a message number is available.
Figure 20.2 Libmaker Help On Window

![Libmaker Interface]

Figure 20.3 Default Configuration Window Toolbar

The Libmaker Default Configuration window tool bar is shown below:

![Default Configuration Window Toolbar]

The Window Tool Bar

The three icons on the left correspond with File menu entries. The next button opens the Online Help dialog. After pressing the Context Help icon (or the shortcut Shift F1), the mouse cursor changes its form and has now an question mark beside the arrow. Then help is called for the next item clicked. You can click on menus, toolbar buttons and on the window area to get specific help.

The command line history contains a list of commands executed. Once you have selected or entered a command in history, clicking Process executes the command. You may use the keyboard shortcut key F2 to jump to the command line. Additionally, there is a context menu associated with the command line:
The Stop icon allows you to stop the current process session.

The next four icons open option settings, the smart slider, the type setting, and message setting dialog box.

The last icon clears the content area.

**Default Configuration Window Status Bar**

Point to a menu entry or icon in the tool bar to display a brief explanation of the button or menu entry in the message area.

**Figure 20.5 Window Status Bar**
Default Configuration Window Menu Bar

File, Libmaker, View and Help options are available from the menu bar.

Figure 20.6 Window Menu Bar

Following functions are available in the menu bar:

Table 20.1 Menu Bar Functions

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Contains entries to manage application configuration files.</td>
</tr>
<tr>
<td>Libmaker</td>
<td>Contains entries to set application options.</td>
</tr>
<tr>
<td>View</td>
<td>Contains entries to customize the application window output.</td>
</tr>
<tr>
<td>Help</td>
<td>A standard Windows Help menu.</td>
</tr>
</tbody>
</table>

Default Configuration Window File Menu

Use the File Menu to save or load configuration files.

Figure 20.7 File Menu

A configuration file contains the following information:

- application option settings specified in the application dialog boxes
- Message settings that specify which messages to display and treat as errors.
- list of last command line executed and current command line.
Libmaker Interface
Libmaker Graphic User Interface

- window position
- Tip of the Day settings

Configuration files are text files with an extension of `.ini`. The user can define as many configuration files as required for the 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 20.2 File Menu Options

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build Library</td>
<td>Opens a standard Open File dialog box. The selected file will be processed as soon as the open File box is closed with OK.</td>
</tr>
<tr>
<td>New/Default Configuration</td>
<td>Resets the application option settings to the default value. The application options that are activated per default are specified in the section Command Line Options.</td>
</tr>
<tr>
<td>Load Configuration</td>
<td>Opens a standard Open File dialog box. Configuration data stored in the selected file is loaded and used by subsequent sessions.</td>
</tr>
<tr>
<td>Save Configuration</td>
<td>Saves the current settings.</td>
</tr>
<tr>
<td>Save Configuration as...</td>
<td>Opens a standard Save As dialog box. Current settings are saved in a configuration file with the specified name.</td>
</tr>
<tr>
<td>Configuration...</td>
<td>Opens the Configuration dialog box to specify the editor used for error feedback and which parts to save with a configuration.</td>
</tr>
<tr>
<td>1..... project.ini</td>
<td>Recent project list. This list can be accessed to open a recently opened project.</td>
</tr>
<tr>
<td>2.....</td>
<td></td>
</tr>
<tr>
<td>Exit</td>
<td>Closes the application.</td>
</tr>
</tbody>
</table>
Default Configuration Libmaker Menu

The Libmaker menu allows you to customize the application. You can set or reset application options or define the optimization level you want to reach.

Figure 20.8 Libmaker Default Configuration Libmaker Menu

Table 20.3 Libmaker Menu Functions

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Options...</td>
<td>Allows you to customize the application. You can set/reset options.</td>
</tr>
<tr>
<td>Messages</td>
<td>Opens a dialog box, where error, warning or information messages can be mapped to another message class (See Message Setting Dialog Box below).</td>
</tr>
<tr>
<td>Stop</td>
<td>Stops the current processing session.</td>
</tr>
</tbody>
</table>
Libmaker Interface
Libmaker Graphic User Interface

Default Configuration Window View Menu

The View menu allows you to customize the application window. You can specify whether the status or tool bar is displayed or hidden. You can also define the font used in the window or clear the window.

Figure 20.9  Libmaker Default Configuration View Menu

Table 20.4  View Menu Functions

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool Bar</td>
<td>hide or show the tool bar in the application window</td>
</tr>
<tr>
<td>Status Bar</td>
<td>hide or show the status bar in the application window</td>
</tr>
<tr>
<td>Log...</td>
<td>allows you to customize the output in the application window content area</td>
</tr>
<tr>
<td>Change Font</td>
<td>opens a standard font selection box; options selected in the font dialog box are applied to the application window content area</td>
</tr>
<tr>
<td>Clear Log</td>
<td>clears the application window content area</td>
</tr>
</tbody>
</table>
Default Configuration Window Help Menu

The Help menu allows you to enable or disable the Tip of the Day dialog, display the help file, and an About box.

Figure 20.10 Libmaker Default Configuration Help Menu

<table>
<thead>
<tr>
<th>Table 20.5 Help Menu Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Menu entry</strong></td>
</tr>
<tr>
<td>Tip of the Day</td>
</tr>
<tr>
<td>Help Topics</td>
</tr>
<tr>
<td>About...</td>
</tr>
</tbody>
</table>

Configuration Window

The three tabs of the Configuration Window let you specify the Editor Settings and the Environment, or Save the Configuration.

Configuration Window Editor Settings Tab

In the Editor Settings tab, select the type of editor to use. Depending on the editor type selected, the tab content changes.
Editor Settings Tab - Global Editor Option

Editor Settings Tab contents when the Global Editor option is chosen are shown in the following figure:

Figure 20.11 Editor Settings Tab - Global Editor Option

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

Editor Settings Tab contents when the Local Editor option is chosen are shown in the following figure:

Figure 20.12 Editor Settings Tab - Local Editor Option

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

The Global and Local Editor configuration can be edited. However, when these entries are stored, the behavior of other tools using the same entry also change when subsequently started.
Editor Settings Tab - Editor Started with Command Line Option

Editor Settings Tab contents when the Editor started with Command Line option is chosen are shown in the following figure:

Figure 20.13 Editor Settings Tab - Editor started with Command Line

When this editor type is selected, a separate editor is associated with the application for error feedback.

Enter the command used to start the editor.

The editor can be started with modifiers. Some Modifiers can be specified on the editor command line that refer to a file name and a line number (See section on Modifiers below).
Examples: (also refer to notes below)

- For IDF use (with path to idf.exe file)
  \texttt{C:\prog\idf.exe %f -g%1,%c}

- For Premia CodeWright V6.0 (with path to cw32.exe file)
  \texttt{C:\Premia\cw32.exe %f -g%1}

- For Winedit 32 bit version use (with path to winedit.exe file)
  \texttt{C:\WinEdit32\WinEdit.exe %f /#:%1}

**Editor Settings Tab - Editor Communication with DDE Option**

Editor Settings Tab contents when the Editor Communication with DDE option is chosen are shown in the following figure:

**Figure 20.14 Editor Settings Tab - Editor Communication with DDE**

![Configuration dialog box with Editor Communication with DDE option selected]

- **Global Editor (Shared by all Tools and all Projects)**
- **Local Editor (Shared by all Tools)**
- **Editor started with Command Line**
- **Editor Communication with DDE**
- **CodeWarrior (with CON)**

<table>
<thead>
<tr>
<th>Service Name</th>
<th>main</th>
<th>system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topic Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client Command</td>
<td>\texttt{[cpp][%2]}</td>
<td></td>
</tr>
</tbody>
</table>

\texttt{use %l for the filename, %d for the line and %c for the column}
Enter the service, topic and client name to be used for a DDE connection to the editor. Entries for Topic and Client Command can have modifiers for file name, line number and column number as explained below.

Examples:

For Microsoft Developer Studio use the following setting:

Service Name : msdev
Topic Name    : system
ClientCommand : [open(\%)\]

UltraEdit is a powerful shareware editor. It is available from www.idmcomp.com or www.ultraedit.com, email idm@idmcomp.com. The latest version of UltraEdit can also be found on the CD-ROM in the 'addons' directory.

For UltraEdit use the following setting:

Service Name : UEDIT32
Topic Name  : system
ClientCommand : [open("%f/%l/%c")]

NOTE The DDE application (Microsoft Developer Studio, UltraEdit) has to be started or else the DDE communication will fail.
Modifiers

The configurations should contain modifiers that 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 message has been detected.
- The %l modifier refers to the line number where the message has been detected.
- The %c modifier refers to the column number where the message has been detected.

**NOTE**
Be careful, the %l modifier can only be used with an editor that can be started with a line number as a parameter. This is not the case for WinEdit version 3.1 or lower, or Notepad. With these editors, you can start with the file name as a parameter and then select the menu entry 'Go to' to jump to the line where the message has been detected. In this case, the editor command looks like:

```
C:\WINAPPS\WINEDIT\Winedit.EXE %f
```

Please check your editor manual to define the command line used to start the editor.
Configuration Window - Save Configuration Tab

The Save Configuration tab of the configuration dialog contains options for the save operation.

Figure 20.16  Configuration Window - Save Configuration Tab

In the Save Configuration tab, selected items can be stored in a project file. This tab has the following items:

- **Options**: If checked, the current option and message settings are saved. Unchecking this option retains the last saved contents.
- **Editor Configuration**: If checked, the current editor settings are saved. Unchecking this option retains the last saved contents.
- **Appearance**: If checked, saves the window position, size, and font used. Also saves the command line content and history in the project file.

**NOTE**  After you have saved the options you want, disable the options that you do not want saved to the “Local Configuration File (Usually project.ini)” in subsequent configuration settings. Uncheck the Save on Exit option to retain settings saved during a previous configuration.
Libmaker Interface
Libmaker Graphic User Interface

- Environment Variables: If checked, environment variables are saved in the project file.
- Save on Exit: If checked, the application writes the configuration settings on exit without confirmation. If not checked, the application does not save configuration changes.

**NOTE**: Almost all settings are stored in the “Local Configuration File (Usually project.ini)”. The only exceptions are:
- The recently used configuration list.
- All settings in this tab.

**NOTE**: Application configuration information can coexist in the same file as the project configuration for the IDF. When an editor is configured by the shell, the application can read this information from the project file, if present. The project configuration file is named project.ini.
Configuration Window - Environment Tab

The Environment tab of the Configuration window is used to configure the environment.

Figure 20.17 Configuration Window - Environment Tab

The content of the dialog is read from the project file in the section [Environment Variables]. Following variables are available:

- General Path: GENPATH
- Object Path: OBJPATH
- Text Path: TEXTPATH
- Absolute Path: ABSPATH
- Header File Path: LIBPATH
- Various Environment Variables: other variables not covered by the above list.
Following command buttons are available:

- Add: Adds a new line/entry
- Change: changes a line/entry
- Delete: deletes a line/entry
- Up: Moves a line/entry up
- Down: Moves a line/entry down

**NOTE** Variables are written to the project file only if you press the Save Button (or select File->Save Configuration, or CTRL-S).
Libmaker Option Settings Window

The Libmaker Option Settings window allows you to set/reset application options.

Figure 20.18 Libmaker Options Settings Window - Messages Tab

Available command line options are displayed in the lower display area. Available options are arranged in different groups. The content of the list box depends on the selected tab, such as Messages (not all groups may be available).
An option is set when its check box is checked. To obtain more detailed information for a specific option, select the option and press the F1 key or help button. To select an option, click the option text. If no option is selected, press F1 or help button to display help for the dialog box.

**NOTE** For options that require additional parameters, an edit box or additional sub window will appear, for example, the option ‘Write statistic output to file...’, in the Output tab.

### Table 20.6 Option Settings Functions

<table>
<thead>
<tr>
<th>Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimization</td>
<td>lists optimization options</td>
</tr>
<tr>
<td>Output</td>
<td>lists options related to output files</td>
</tr>
<tr>
<td>Input</td>
<td>lists options related to input file.</td>
</tr>
<tr>
<td>Language</td>
<td>lists options related to programming language (ANSI C, C++)</td>
</tr>
<tr>
<td>Target</td>
<td>lists options related to target processor</td>
</tr>
<tr>
<td>Host</td>
<td>lists options related to the host</td>
</tr>
<tr>
<td>Code Generation</td>
<td>lists options related to code generation (memory models, float format,...)</td>
</tr>
<tr>
<td>Messages</td>
<td>lists options that control generation of error messages</td>
</tr>
<tr>
<td>Various</td>
<td>lists options not related to the above</td>
</tr>
</tbody>
</table>
Libmaker Interface
Libmaker Graphic User Interface

Libmaker Message Settings Window

This window allows you to map messages to a different message class. A tab is available for each error message group: Disabled, Information, Warning, Error and Fatal.

Each message has its own character (e.g. ‘C’ for Compiler messages, ‘A’ for Assembler messages, ‘L’ for Linker messages, ‘M’ for Maker messages, ‘LM’ for Libmaker messages) followed by a 4-5 digit number.

Figure 20.19 Libmaker Message Settings Window

In this window, some command buttons may be disabled. For example, if a message cannot be mapped as an Information message, the “Move to” group ‘Information’ command button is disabled when this message is highlighted.
Table 20.7 Message Classes

<table>
<thead>
<tr>
<th>Message group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disabled</td>
<td>Lists all disabled messages that will not be displayed by the application.</td>
</tr>
<tr>
<td>Information</td>
<td>Lists all information messages.</td>
</tr>
<tr>
<td>Warning</td>
<td>Lists all warning messages. Input file processing continues, if a warning occurs.</td>
</tr>
<tr>
<td>Error</td>
<td>Lists all error messages. Input file processing continues, if an error occurs.</td>
</tr>
<tr>
<td>Fatal</td>
<td>Lists all fatal error messages. If a fatal message occurs, processing stops immediately. Fatal messages can not be changed.</td>
</tr>
</tbody>
</table>

Table 20.8 Command Button Functions

<table>
<thead>
<tr>
<th>Command Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move to: Disabled</td>
<td>Selected messages will be disabled.</td>
</tr>
<tr>
<td>Move to: Information</td>
<td>Selected messages will be information messages.</td>
</tr>
<tr>
<td>Move to: Warning</td>
<td>Selected messages will be warning messages.</td>
</tr>
<tr>
<td>Move to: Error</td>
<td>Selected messages will be error messages.</td>
</tr>
<tr>
<td>Move to: Default</td>
<td>Selected messages will revert back to their default mapping.</td>
</tr>
<tr>
<td>Reset All</td>
<td>Resets all messages to their default.</td>
</tr>
<tr>
<td>Ok</td>
<td>Exits and accepts changes.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Exits without accepting changes.</td>
</tr>
<tr>
<td>Help</td>
<td>Displays online help.</td>
</tr>
</tbody>
</table>
Changing the Class Associated with a Message

You can configure your own message mapping. You can use the buttons located on the right side of the dialog box. Each button refers to a message class. To change the class associated with a message, select the message in the list box and then click the button associated with another class.

**NOTE**  The ‘move to’ buttons are only active for messages that can be moved.

Example:
To change a warning message to an error message:
- Click the Warning tab to display the list of all warning messages.
- Click on the message you want to change.
- Click Error to define this message as an error message.

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

If you want to validate the new error message mapping, click OK to close the 'Message settings' dialog box. If you click 'Cancel', the previous message mappings remain valid.

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 click Help or the F1 key. An information box appears, which contains a detailed description of the error message and an example of code that produces the message. If several messages are selected, help for the first message is shown. If no message is selected, pressing the F1 key or help button displays help for this dialog box.
About Libmaker Dialog Box

Select Help->about to display the about box. The about box contains the current directory and version information for application modules. The main version is displayed at the top of the dialog.

The ‘Extended Information’ button displays license information about all software components in the same directory as the executable. Click on OK to close this dialog.

NOTE During processing, subversions of the application modules can not be requested. They are only displayed when the application is not processing information.

Figure 20.20 About Libmaker Dialog Box
Libmaker Interface
Libmaker Environment Variables

Libmaker Environment Variables

This section describes the environment variables used by the Libmaker utility. Some of the environment variables are also used by other tools (e.g. Linker/Assembler).

There are three ways to specify environment variables:

- In the section [Environment Variables] in the current project file. This file may be specified at Tool startup with the "-Prod: Specify Project File at Startup" option. This is recommended and also supported by the IDF.
- An optional 'default.env' file in the current directory. This file is supported for compatibility with earlier versions. This file may be specified with the variable "ENVIRONMENT: Environment File Specification". Using the default.env file is not recommended.
- Setting environment variables at the system level (DOS level). This is not recommended.

Parameters may be set in an environment using environment variables. The syntax is:
Parameter = KeyName "=" ParamDef.

**NOTE**
Normally 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 definitions in the project file in the section named [Environment Variables].
- Putting definitions in a file called DEFAULT.ENV (.hidefaults for UNIX) in the default directory.

**NOTE**
The maximum length of environment variables in the DEFAULT.ENV/.hidefaults file is 65535 characters.

- Putting definitions in a file given by the value of the system environment variable ENVIRONMENT.

**NOTE**
The default directory mentioned above can be set via the system environment variable "DEFAULTDIR: Current Directory".
All programs first search the system environment for environment variables, 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: Set Environment
Variable" option. Ensure that no spaces exist at the end of environment
variables.

### Local Configuration File (Usually project.ini)

The Libmaker utility usually uses either the default.env configuration file or the
project.ini (default) configuration file.

### The Current Directory

The current directory is the most important environment for all tools. The current
directory is the base search directory where the tool searches for files (e.g. for
DEFAULT.ENV/.hidefaults)

Normally, the current directory of a tool is determined by the operating system or program
that launches another one (e.g. IDF).

For the UNIX operating system, the current directory is also the current directory where
the binary file was 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 one
  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 tool with its own current directory (e.g. the IDF
  editor), the current directory is the one specified by the launching tool (e.g. current
  directory defined for IDF).
- For tools, the current directory is the directory containing the local project file.
  Changing the current project file also changes the current directory, if the other
  project file is in a different directory. Note that browsing for a C file does not change
  the current directory.
Libmaker Interface

Libmaker Environment Variables

To overwrite this behavior, the environment variable “DEFAULTDIR: Current Directory” may be used.

The current directory is displayed with other information with the option “-V: Prints the Libmaker Version”.

Paths

Most environment variables contain path lists indicating 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:\PROJECTS\TESTS;/usr/local/lib;/home/me/my_project

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

Example:

LIBPATH=*C:\INSTALL\LIB

NOTE Some DOS/UNIX environment variables (like GENPATH, LIBPATH, etc.) are used.

We strongly recommend using the IDF and setting the environment by means of a <project>.pjt (.hidefaults for UNIX) file in your project directory. This way, you can have different projects in different directories, each with its own environment.

NOTE When using WinEdit, do not set the system environment variable “DEFAULTDIR: Current Directory”. If you do and this variable does not contain the project directory given in WinEdit’s project configuration, files might not be placed where you expect them.
Line Continuation

It is possible to specify an environment variable in an environment file (default.env/.hidefaults) over multiple lines using the line continuation character `\`:

Example:
```plaintext
OPTIONS=
  -W2 \ -Wpd
```

This is the same as
```plaintext
OPTIONS=-W2 -Wpd
```

Be careful using this feature with paths, e.g.
```plaintext
GENPATH=.
TEXTFILE=.	xt
```

will result in
```plaintext
GENPATH=.
TEXTFILE=.	xt
```

To avoid such problems, we recommend using a semicolon `;` at the end of a path, if there is a `\` at the end:
```plaintext
GENPATH=.
TEXTFILE=.	xt
```
Environment Variable Details

This section describes each available environment variable. Variables are listed in alphabetical order and described in the following paragraphs.

Table 20.9 Environment Variable Details

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tools</td>
<td>Lists tools that use this variable.</td>
</tr>
<tr>
<td>Synonym</td>
<td>For some environment variables, a synonym also exists. Synonyms may be used for earlier releases of the Decoder and will be removed in the future. A synonym has lower precedence than the environment variable.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Specifies the syntax of the option in EBNF format.</td>
</tr>
<tr>
<td>Arguments</td>
<td>Describes and lists optional and required arguments for the variable.</td>
</tr>
<tr>
<td>Default</td>
<td>Shows the default setting for the variable, if one exists.</td>
</tr>
<tr>
<td>Description</td>
<td>Provides a detailed description of the option and how to use it.</td>
</tr>
<tr>
<td>Example</td>
<td>Gives an example of usage, and effects of the variable when possible. Shows an entry in the default.env for PC or in the .hidefaults for UNIX.</td>
</tr>
<tr>
<td>See also</td>
<td>Names related sections.</td>
</tr>
</tbody>
</table>

DEFAULTDIR: Current Directory

Tools:
 Compiler, Assembler, Linker, Decoder, Debugger, Libmaker, Maker

Synonym:
 None.

Syntax:
 "DEFAULTDIR=" <directory>.
Libmaker Interface

Arguments:

<directory>: Specify the default directory.

Default:

None.

Description:

With this environment variable, specify the default directory for all tools. All tools indicated above will use the directory specified as their current directory instead of the one defined by the operating system or launching tool (e.g. editor).

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

Example:

DEFAULTDIR=C:\INSTALL\PROJECT

ENVIRONMENT: Environment File Specification

Tools:

Compiler, Linker, Decoder, Debugger, Libmaker, Maker

Synonym:

HIENVIRONMENT

Syntax:

"ENVIRONMENT= " <file>.

Arguments:

<file>: file name and pat, without spaces

Default:

DEFAULT.ENV on PC, .hidefaults on UNIX

Description:

This variable is specified at the system level (global). Usually the Decoder looks in the current directory for an environment file named DEFAULT.ENV (.hidefaults on
Libmaker Interface

Libmaker Environment Variables

UNIX). Using `ENVIRONMENT` a different file name may be specified (e.g. in the `AUTOEXEC.BAT` (DOS) or `.cshrc` (UNIX) file).

**NOTE** This is a system level (global) environment variable. It cannot be specified in a default environment file (`DEFAULT.ENV/.defaults`).

**Example:**

```
ENVIRONMENT=\Freescale\prog\global.env
```

**ERRORFILE: Error File Name Specification**

**Tools:**

Compiler, Assembler, Linker, Burner, Libmaker

**Synonym:**

None

**Syntax:**

```
"ERRORFILE= " <file name>
```

**Arguments:**

`<file name>`: File name with possible format specifiers

**Description:**

The `ERRORFILE` environment variable specifies the name for the error file. 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`).

A notification box is shown in the event of an invalid error file name.

**Example:**

```
ERRORFILE=MyErrors.err
```
Lists all errors into the file MyErrors.err in the current 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. If you compile a file such as \sources\test.c, an error list file, \sources\test.err, is generated.

ERRORFILE=\dir1\%n.err

For a source file such as test.c, an error list file \dir1\test.err is generated.

ERRORFILE=%p\errors.txt

For a source file such as \dir1\dir2\test.c, an error list file \dir1\dir2\errors.txt is generated.

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

Example:

Another example shows the usage of this variable to support correct error feedback with the WinEdit Editor. The editor looks for an error file named EDOUT, as shown:

Installation directory: E:\INSTALL\PROG
Project sources: D:\MEPHISTO
Common Sources for projects: E:\CLIB

Entry in default.env (D:\MEPHISTO\DEFAULT.ENV):
ERRORFILE=E:\INSTALL\PROG\EDOUT

Entry in WINEDIT.INI (in Windows directory):
OUTPUT=E:\INSTALL\PROG\EDOUT

NOTE Be careful to set this variable if the WinEdit Editor is use, otherwise the editor cannot find the EDOUT file.
Libmaker Interface
Libmaker Environment Variables

GENPATH: Defines Paths to search for input Files

Tools:
   Compiler, Assembler, Linker, Decoder, Debugger

Synonym:
   HIPATH

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

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

Description:
Libmaker will look for the required input files (binary input files and source files) in the project directory, then in the directories listed in the environment variable GENPATH.

NOTE  If a directory specification in this environment variable starts with an asterisk (**), the directory tree is searched recursively, i.e. all subdirectories are also searched.

Example:
   GENPATH=\obj;..\.\lib;
**TEXTPATH: Text Path**

**Tools:**
Compiler, Assembler, Linker, Decoder

**Synonym:**
None

**Syntax:**
"TEXTPATH=" \{<path>\).

**Arguments:**

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

**Description:**
When this environment variable is defined, Libmaker will store the list file produced in the first directory specified. If `TEXTPATH` is not set, the generated `.LST` file will be stored in the directory containing the binary input file.

**Example:**

```
TEXTPATH=\sources ..\..\headers;\usr\local\txt
```

---

**TMP: Temporary Directory**

**Tools:**
Compiler, Assembler, Linker, Debugger, Libmaker

**Synonym:**
None.

**Syntax:**
"TMP=" \<directory>.

**Arguments:**

- `<directory>`: Directory used for temporary files.
Libmaker Interface
Libmaker Environment Variables

Default:
None.

Description:
If a temporary file is needed, normally the ANSI function tmpnam() is used. This library function specifies the directory to store temporary files. 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 a system level (global) environment variable. It CANNOT be specified in a default environment file (DEFAULT.ENV/.hidefaults).

Example:
TMP=C:\TEMP

See also:
• “The Current Directory”
Libmaker Options

Options

Libmaker offers a number of options to control operation. Options are composed of a minus/dash ( - ) followed by one or more letters or digits. Anything not starting with a dash/minus is considered to be a parameter file to be linked. Options can be specified on the command line.

NOTE  Arguments for an option must not exceed 128 characters.

Command line options are not case sensitive, e.g. -otest.lst is the same as -OTEST.LST.

Option Details

This section describes each of the available Libmaker options. The options are listed in alphabetical order and described in the format below.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group (Option Groups)</td>
<td>OUTPUT: Options that control the format and content of the list file.</td>
</tr>
<tr>
<td></td>
<td>INPUT: Options that control and specify input files</td>
</tr>
<tr>
<td></td>
<td>HOST: Host and Operating System dependent options</td>
</tr>
<tr>
<td></td>
<td>MESSAGE: Options that control the error and message output</td>
</tr>
<tr>
<td>Syntax</td>
<td>Specifies the syntax of the option in EBNF format.</td>
</tr>
<tr>
<td>Arguments</td>
<td>Describes and lists optional and required arguments for the option.</td>
</tr>
<tr>
<td>Default</td>
<td>Default setting for the option.</td>
</tr>
<tr>
<td>Description</td>
<td>Provides a detailed description of the option and how to use it.</td>
</tr>
<tr>
<td>See also</td>
<td>Related options.</td>
</tr>
</tbody>
</table>
Using Special Modifiers

With some options, it is possible to use special modifiers. However, some modifiers may not make sense for all options. This section describes the modifiers that are supported:

Table 21.2 Special Modifiers

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%p</td>
<td>path including file separator</td>
</tr>
<tr>
<td>%N</td>
<td>file name in strict 8.3 format</td>
</tr>
<tr>
<td>%n</td>
<td>file name without extension</td>
</tr>
<tr>
<td>%E</td>
<td>extension in strict 8.3 format</td>
</tr>
<tr>
<td>%e</td>
<td>extension</td>
</tr>
<tr>
<td>%f</td>
<td>path + file name without extension</td>
</tr>
<tr>
<td>%&quot;</td>
<td>a double quote (&quot;) if the file name, path or extension contains a space</td>
</tr>
<tr>
<td>%'</td>
<td>a single quote (') if the file name, path or extension contains a space</td>
</tr>
<tr>
<td>%%</td>
<td>replaced with contents of an environment variable</td>
</tr>
<tr>
<td>%%</td>
<td>generates a single '%' character</td>
</tr>
</tbody>
</table>

Examples

The examples assume that the base file name for modifiers is:

`c:\Freescale\my demo\TheWholeThing.myExt`

%p gives the path only with a file separator:

`c:\Freescale\my demo\`

%N truncates the file name to 8.3 format, 8 characters:

`TheWhole`

%n returns the file name without extension:

`TheWholeThing`

%E limits the extension to 8.3 format, only 3 characters:

`myE`

%e is used for the complete extension:

`myExt`
%f gives the path plus the file name:

c:\Freescale\my demo\TheWholeThing

Because the path contains a space, using %" or %' is recommended: Thus %"%f%"
gives:

"c:\Freescale\my demo\TheWholeThing"

where %\f\% gives:

'c:\Freescale\my demo\TheWholeThing'

Using %\(envVariable\) an environment variable may also be used. A subsequent file
separator after %\(envVariable\) is ignored, if the environment variable is empty or does not
exist. For example, $(TEXTPATH)\myfile.txt is replaced with:

c:\Freescale\txt\myfile.txt

if TEXTPATH is set to:

TEXTPATH=c:\Freescale\txt

But is set to:

myfile.txt

if TEXTPATH does not exist or is empty.

%% may be used to print a percent sign. %e%% gives:

myExt%
Libmaker Options

Options

Description:
You can either run a libmaker command file (preceded by ‘@‘), or use the -Cmd command on the command line to run libmaker commands. Alternatively, you can use the command without the ‘+’ operator as well:

-Cmd"a.o b.o c.o = d.lib"

Instead of “..” to wrap around the command string, you can use as well:

-Cmd(a.o b.o c.o = d.lib)
-Cmd[a.o b.o c.o = d.lib]
-Cmd(a.o b.o c.o = d.lib)
-Cmd’a.o b.o c.o = d.lib’

If your file names have spaces or operator characters in the file name, you need to use double quotes for the file name:

-Cmd(a.o "my b.o" "c-c.o" = d.lib)

You still can use double quotes for the -Cmd option, but in such a case you need to double-double quote files names in double quotes:

-Cmd"a.o ""my b.o"" ""c-c.o"" = d.lib"

Example:

-Cmd"a.o + b.o = c.lib"

See also:

- “-Mar: Freescale Archive Commands”

-Env: Set Environment Variable

Group:
HOST

Syntax:

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

Arguments:

<Environment Variable>: Environment variable to be set
<Variable Setting>: Value of environment variable
Default:
None.

Description:
This option sets an environment variable. This environment variable may be used in the maker or used to overwrite system environment variables.

Example:
-EnvOBJPATH=\sources\obj
This is the same as
OBJPATH=\sources\obj
in the default.env.
To use an environment variable with file names that contain spaces, use the following syntax:
-Env"OBJPATH=\program files"

See also:
- "Libmaker Environment Variables"

-H: Short Help

Group:
VARIOUS

Scope:
None

Syntax:
-H

Arguments:
None

Default:
None
**Libmaker Options**

*Options*

**Description:**
The `-H` option causes the tool to display a short list (i.e. help list) of available options within the output window.
No other option or source file should be specified when the `-H` option is invoked.

**Example:**

```bash
-H may produce following list:
```

<table>
<thead>
<tr>
<th>HOST:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-Env</code></td>
<td>Set environment variable</td>
</tr>
<tr>
<td><code>-View</code></td>
<td>Application Standard Occurrence</td>
</tr>
<tr>
<td></td>
<td><code>-ViewWindow Window</code></td>
</tr>
<tr>
<td></td>
<td><code>-ViewMin Min</code></td>
</tr>
<tr>
<td></td>
<td><code>-ViewMax Max</code></td>
</tr>
<tr>
<td></td>
<td><code>-ViewHidden Hidden</code></td>
</tr>
</tbody>
</table>

---

**-Lic: License Information**

**Group:**

VARIOUS

**Scope:**

None

**Syntax:**

```
-Lic
```

**Arguments:**

None

**Default:**

None

**Description:**
The `-Lic` option prints the current license information (e.g. if it is a demo version or a full version). This information is also displayed in the About box.
-LicA: License Information About Every Feature in Directory

Group:
VARIABLE

Scope:
None

Syntax:
-LicA

Arguments:
None

Default:
None

Description:
The -LicA option prints the license information (e.g. if the tool or feature is a demo version or a full version) of every tool or .dll in the directory where the executable is located. This will take some time as every file in the directory is analyzed.

Example:
-LicA

See also:
• “-Lic: License Information”
• “-LicBorrow: Borrow License Feature”
-LicBorrow: Borrow License Feature

**Group:**
HOST

**Scope:**
None

**Syntax:**
"-LicBorrow"<feature>[";"<version>]";"<Date>

**Arguments:**
- `<feature>`: the feature name to be borrowed (e.g. HI100100).
- `<version>`: optional version of the feature to be borrowed (e.g. 3.000).
- `<date>`: date with optional time until when the feature shall be borrowed (e.g. 15-Mar-2004:18:35).

**Default:**
None

**Description:**
This option allows you to borrow a license feature until a given date/time. Borrowing allows you to use a floating license even if disconnected from the floating license server.

You need to specify the feature name and the date until you want to borrow the feature. If the feature you want to borrow is a feature belonging to the tool where you use this option, then you do not need to specify the version of the feature (because the tool knows the version). However, if you want to borrow any feature, you need to specify as well the feature version of it.

You can check the status of currently borrowed features in the tool about box.

**NOTE**
You can only borrow features if you have a floating license and if your floating license is enabled for borrowing. See as well the provided FLEXlm documentation about details on borrowing.

**Example:**
- LicBorrowHI100100;3.000;12-Mar-2004:18:25
-LicWait: Wait Until Floating License Available from Floating License Server

Group: HOST

Scope: None

Syntax: 

"-LicWait"

Arguments: None

Default: None

Description:
By default, if a license is not available from the floating license server, then the application returns immediately. With -LicWait set, the application waits (blocking) until a license is available from the floating license server.

Example:

-LicWait

See also:

- “-Lic: License Information”
- “-LicA: License Information About Every Feature in Directory”
- “-LicBorrow: Borrow License Feature”
Libmaker Options

Options

-Mar: Freescale Archive Commands

Group:

OUTPUT

Syntax:

"-Mar" "" <library> [ <member> ] "".

Arguments:

<library>: name of the library.
<member>: list of members for the library to be added.

Default:

None.

Description:

This command provides a more 'ar' (archive) like way to create a library out of object files. Instead of

-Cmd"a.o b.o c.o = d.lib"

this can be done with

-Mar"d.lib a.o b.o c.o"

Unlike the -Cmd command, no operator processing (+/-) is done, so this makes it easier to deal with file names having operator characters in it.

Example:

-Mar"c.lib a.o b.o"

See also:

- "-Cmd: Libmaker Commands"

-N: Display Notify Box (PC Only)

Group:

MESSAGE

460 HC(S)08/RS08 and HC(S)12 Build Tools Utilities Manual
Syntax:

“-N”.

Arguments:

None.

Default:

None.

Description:

The Compiler displays an alert box if an error occurs during compilation. This is useful when running a make file (See: Make Utility), since the Compiler waits for the user to acknowledge the message, thus suspending make file processing. (The ‘N’ stands for “Notify”.)

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

Example:

- N

If an error occurs during compilation, a dialog box similar to the following will appear:

Figure 21.1 Libmaker Error Notice

-NoBeep: No Beep in Case of an Error

Group:

MESSAGE
Libmaker Options

Options

Syntax:

"-NoBeep".

Arguments:

None.

Default:

None.

Description:

Normally there is a ‘beep’ at the end of processing, if an error occurs. This option disables the ‘beep’.

Example:

-NoBeep

-NoPath: Strip Path Info

Group:

OUTPUT

Syntax:

"-NoPath".

Arguments:

None.

Default:

None.

Description:

With this option it is possible to avoid path information in object files. This is useful if you want to move object files to another file location or hide your path structure.

Example:

-NoPath
-Prod: Specify Project File at Startup

**Group:**
Startup - This option cannot 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 application. It cannot be specified in any other circumstances, including the `default.env` file, the command line, etc.

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

**Example:**
```
compiler.exe -prod=project.ini
```
Use the compiler executable name instead of “compiler”.

**See also:**
- “Local Configuration File (Usually project.ini)”

-V: Prints the Libmaker Version

**Group:**
VARIOUS
Libmaker Options

Options

Syntax:

"-V"

Arguments:
None

Default:
None

Description:
This option prints the Compiler version of the internal subversion numbers of the parts the Compiler consists of and the current directory.

NOTE
This option can be used to determine the current directory.

Example:

-V produces the following list:

Libmaker

Directory: c:\test
Project file: c:\test\project.ini
GENPATH=
LIBPATH=
OBJPATH=c:\test
ABSPATH=c:\test
TEXTPATH=c:\test

Common Module V-5.0.7, Date Feb 8 2002
User Interface Module, V-5.0.23, Date Feb 8 2002
Libmaker Module V-5.0.5, Date Feb 13 2002

-View Application Standard Occurrence (PC Only)

Group:

HOST
Syntax: 

"-View" <kind>.

Arguments: 

<kind> is one of:
Window: Default window size for application
Min: Application window is minimized
Max: Application window is maximized
Hidden: Application window is not visible

Default: 
Application started with arguments: Minimized.
Application started without arguments: Window.

Description:
Normally the application (e.g. linker, compiler,...) is started in a 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 minimized to allow batch processing. However, with this option the behavior may be specified.

Using -ViewWindow, the application is visible in its normal window. Using -ViewMin, the application is iconified (in the task bar). Using -ViewMax, the application is maximized.

Using -ViewHidden, the application processes arguments (e.g. files to be compiled/linked) in the background. However, if you use the "-N: Display Notify Box (PC Only)" option, a dialog box is still possible.

Example:

c:\Freescale\libmaker.exe -ViewHidden -Cmd"a.o b.o = c.lib"

-W1: No Information Messages

Group:
MESSAGE
Libmaker Options

Options

Syntax:

"-W1".

Arguments:
None.

Default:
None.

Description:
This option suppresses INFORMATION messages. Only WARNING and ERROR messages are generated.

Example:

-W1

See also:

- "-WmsgNi: Number of Information Messages"

-W2: No Information and Warning Messages

Group:
MESSAGE

Syntax:

"-W2".

Arguments:
None.

Default:
None.

Description:
Suppresses all INFORMATION and WARNING messages, only ERRORs are generated.
Example:

-W2

See also:

- \"-WmsgNi: Number of Information Messages\"
- \"-WmsgNw: Number of Warning Messages\"

-Wmsg8x3: Cut File Names in Microsoft Format to 8.3 (PC Only)

Group:

MESSAGE

Syntax:

\"-Wmsg8x3\".

Arguments:

None.

Default:

None.

Description:

Some editors, like the early versions of WinEdit, expect the file name to be in a strict 8.3 format: a maximum of eight characters with a three character extension. In Win95 or WinNT, longer filenames are possible. This option truncates the file name to the 8.3 format.

Example:

x:\mysourcefile.c(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:

- \"-WmsgFim (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode\"
- \"-WmsgFbm (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode\"
-WErrFile: Create “err.log” Error File

Group:
   MESSAGE

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

Arguments:
   None.

Default:
   The err.log is created/deleted.

Description:
   A return code is used for error feedback to called tools. In a 16 bit windows environment, this was not possible. A file err.log with error numbers was used to signal an error. To ignore errors, the file err.log was deleted. With UNIX or WIN32, a return code is now available, so this file is no longer needed if UNIX / WIN32 applications are involved. To use a 16-bit maker with this tool, the error file must be created in order to signal an error.

Example:

- WErrFileOn
  err.log is created/deleted when the application is finished.
- WErrFileOff
  existing err.log is not modified.

See also:

* “-WStdout: Write to Standard Output”
* “-WOutFile: Create Error List File”
-WmsgCE: RGB Color for Error Messages

Group:
MESSAGE

Syntax:
"-WmsgCE" <RGB>

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

Default:
-WmsgCE16711680 (rFF g00 b00, red)

Description:
This option changes the error message color. The specified value must be an RGB (Red-Green-Blue) value, and may be specified in decimal. Hexadecimal needs a 0X prefix, (for gray errors: -WmsgCE0x808080).

Example:
-WmsgCE255 changes the color of error messages to blue

-WmsgCF: RGB Color for Fatal Messages

Group:
MESSAGE

Syntax:
"-WmsgCF" <RGB>

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

Default:
-WmsgCF8388608 (r80 g00 b00, dark red)
**Libmaker Options**

*Options*

**Description:**
This option changes the color of a fatal message. The specified value must be an RGB (Red-Green-Blue) value, and may be specified in decimal. Hexadecimal needs a 0X prefix, (for gray errors: `-WmsgCF0x808080`).

**Example:**
- `-WmsgCF255` changes the color of fatal messages to blue

---

**-WmsgCI: RGB Color for Information Messages**

**Group:**
MESSAGE

**Syntax:**
```
"-WmsgCI" <RGB>
```

**Arguments:**

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

**Default:**
- `-WmsgCI32768` (r00 g80 b00, green)

**Description:**
This option changes the color of an information message. The specified value must be an RGB (Red-Green-Blue) value, and may be specified in decimal. Hexadecimal needs a 0X prefix, (for gray errors: `-WmsgCI0x808080`).

**Example:**
- `-WmsgCI255` changes the information messages to blue
-**WmsgCU**: RGB Color for User Messages

**Group:**
MESSAGE

**Syntax:**

```
"-WmsgCU" <RGB>
```

**Arguments:**

- `<RGB>`: 24bit RGB (red green blue) value

**Default:**

`-WmsgCU0` (r00 g00 b00, black)

**Description:**
This option changes the color of a user message. The specified value must be an RGB (Red-Green-Blue) value, and may be specified in decimal. Hexadecimal needs a 0X prefix, (for gray errors: `-WmsgCU0x808080`).

**Example:**

`-WmsgCU255` changes the user messages to blue

---

-**WmsgCW**: RGB Color for Warning Messages

**Group:**
MESSAGE

**Syntax:**

```
"-WmsgCW" <RGB>
```

**Arguments:**

- `<RGB>`: 24bit RGB (red green blue) value

**Default:**

`-WmsgCW255` (r00 g00 bFF, blue)
Libmaker Options
Options

Description:
This option changes the color of a warning message. The specified value must be an RGB (Red-Green-Blue) value, and may be specified in decimal. Hexadecimal needs a 0X prefix, (for gray errors: \texttt{-WmsgCw0x808080}).

Example:
\texttt{-WmsgCw0} changes the warning messages to black

-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode

Group:
MESSAGE

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

Arguments:
ν : Verbose format.
m : Microsoft format.

Default:
-WmsgFbm

Description:
The Compiler can be started with additional arguments, such as files to be compiled, together with Compiler options. If the Compiler has been started with arguments (e.g. from the Make Tool or with the “%f” argument from the IDF), the Compiler compiles the files in batch mode, no Compiler window is visible and the Compiler terminates after job completion.

If the compiler is in batch mode, compiler messages are written to a file instead of the screen. By default, the Compiler uses the Microsoft message format to write Compiler messages (errors, warnings, information messages), if the compiler is in batch mode.

With this option, the default format may be changed from the Microsoft format (line information only) to a more verbose error format with line, column and source information.
NOTE Using the verbose message format may slow down compilation, because the compiler has to write more information into the message file.

Example:

```c
void fun(void) {
   int i, j;
   for(i=0;i<1;i++);
}
```

By default, the Compiler will produce the following file, if running in batch mode (e.g. started from the Make tool):

```
X:\C.C(3): INFORMATION C2901: Unrolling loop
X:\C.C(2): INFORMATION C5702: j: declared in function fun but not referenced
```

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

```
-WmsgFbv
>> in "X:\C.C", line 3, col 2, pos 33
   int i, j;
   for(i=0;i<1;i++);
^  
INFORMATION C2901: Unrolling loop
>> in "X:\C.C", line 2, col 10, pos 28
void fun(void) {
   int i, j;
^  
INFORMATION C5702: j: declared in function fun but not referenced
```

See also:

- “ERRORFILE: Error File Name Specification”
- “-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode”
-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode

**Group:**
MESSAGE

**Syntax:**
```
"-WmsgFi" \["v" | "m"]
```

**Arguments:**

v : Verbose format.
m : Microsoft format.

**Default:**
-WmsgFiv

**Description:**
If the Compiler is started without additional arguments such as files compiled, along with Compiler options, the Compiler is in interactive mode (GUI window is visible).

By default, the Compiler uses the verbose error file format to write Compiler 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).

**NOTE**
Using the Microsoft format may speed up compilation, because the compiler writes less information to the screen.

**Example:**

```c
void fun(void) {
    int i, j;
    for(i=0;i<1;i++);
}
```
By default, the Compiler may produce the following error output in the Compiler window, if running in interactive mode:

Top: X:\C.C
Object File: X:\C.O

>> in "X:\C.C", line 3, col 2, pos 33
int i, j;

    for(i=0;i<1;i++);

^  
INFORMATION C2901: Unrolling loop

Set the format to Microsoft to display less information:

-WmsgFim
Top: X:\C.C
Object File: X:\C.O
X:\C.C(3): INFORMATION C2901: Unrolling loop

See also:

* "ERRORFILE: Error File Name Specification"
* "-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode"

-WmsgFob: Message Format for Batch Mode

Group:
MESSAGE

Syntax:
-``-WmsgFob"<string>". 

Arguments:
<string>: format string.

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

Options

**Description:**

Use this option to modify the default message format in batch mode. Following formats are supported (example source file is `x:\Freescale\mysourcefile.cpph`).

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%s</td>
<td>Source Extract</td>
<td><code>x:\Freescale\</code></td>
</tr>
<tr>
<td>%p</td>
<td>Path</td>
<td><code>x:\Freescale\mysourcefile</code></td>
</tr>
<tr>
<td>%f</td>
<td>Path and name</td>
<td><code>x:\Freescale\mysourcefile</code></td>
</tr>
<tr>
<td>%n</td>
<td>File name</td>
<td><code>mysourcefile</code></td>
</tr>
<tr>
<td>%e</td>
<td>Extension</td>
<td><code>.cpph</code></td>
</tr>
<tr>
<td>%N</td>
<td>File (8 chars)</td>
<td><code>mysource</code></td>
</tr>
<tr>
<td>%E</td>
<td>Extension (3 chars)</td>
<td><code>.cpp</code></td>
</tr>
<tr>
<td>%l</td>
<td>Line</td>
<td><code>3</code></td>
</tr>
<tr>
<td>%c</td>
<td>Column</td>
<td><code>47</code></td>
</tr>
<tr>
<td>%o</td>
<td>Pos</td>
<td><code>1234</code></td>
</tr>
<tr>
<td>%K</td>
<td>Uppercase kind</td>
<td><code>ERROR</code></td>
</tr>
<tr>
<td>%k</td>
<td>Lowercase kind</td>
<td><code>error</code></td>
</tr>
<tr>
<td>%d</td>
<td>Number</td>
<td><code>C1815</code></td>
</tr>
<tr>
<td>%m</td>
<td>Message</td>
<td><code>text</code></td>
</tr>
<tr>
<td>%</td>
<td>Percent</td>
<td><code>%</code></td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
<td></td>
</tr>
<tr>
<td>%*</td>
<td>A * if the filename,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if the path or the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>extension contains</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a space</td>
<td></td>
</tr>
<tr>
<td>%'</td>
<td>A ' if the filename,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>the path or the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>extension contains</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a space</td>
<td></td>
</tr>
</tbody>
</table>

**Example:**

```bash
-WmsgFob "%f%e(%l): %k %d: %m
"
```

produces a message in following format:

`x:\C.C(3): information C2901: Unrolling loop`

**See also:**

- “ERRORFILE: Error File Name Specification”
- “-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode”
- “-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode”
- “-WmsgFonp: Message Format for No Position Information”
- “-WmsgFoi: Message Format for Interactive Mode”
-WmsgFoi: Message Format for Interactive Mode

Group: MESSAGE

Syntax: "-WmsgFoi"<string>.

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

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

Description:
Use this option to modify the default message format in interactive mode. The following formats are supported (example source file is x:\Freescale\mysourcefile.cpph):

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%s</td>
<td>Source Extract</td>
<td>x:\sources\</td>
</tr>
<tr>
<td>%p</td>
<td>Path</td>
<td>x:\sources\mysourcefile</td>
</tr>
<tr>
<td>%f</td>
<td>Path and name</td>
<td>x:\sources\mysourcefile</td>
</tr>
<tr>
<td>%n</td>
<td>File name</td>
<td>mysourcefile</td>
</tr>
<tr>
<td>%e</td>
<td>Extension</td>
<td>.cpph</td>
</tr>
<tr>
<td>%N</td>
<td>File (8 chars)</td>
<td>mysource</td>
</tr>
<tr>
<td>%E</td>
<td>Extension (3 chars)</td>
<td>.cpp</td>
</tr>
<tr>
<td>%l</td>
<td>Line</td>
<td>3</td>
</tr>
<tr>
<td>%c</td>
<td>Column</td>
<td>47</td>
</tr>
<tr>
<td>%o</td>
<td>Pos</td>
<td>1234</td>
</tr>
<tr>
<td>%K</td>
<td>Uppercase kind</td>
<td>ERROR</td>
</tr>
<tr>
<td>%k</td>
<td>Lowercase kind</td>
<td>error</td>
</tr>
<tr>
<td>%d</td>
<td>Number</td>
<td>C1815</td>
</tr>
<tr>
<td>%m</td>
<td>Message</td>
<td>text</td>
</tr>
<tr>
<td>%%</td>
<td>Percent</td>
<td>%</td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
<td></td>
</tr>
<tr>
<td>%*</td>
<td>A &quot; if the filename, if the path or the extension contains a space</td>
<td></td>
</tr>
</tbody>
</table>

HC(S)08/RS08 and HC(S)12 Build Tools Utilities Manual
Libmaker Options
Options

`%' A ' if the filename, the path or the extension contains a space

Example:

`-WmsgFoi"%f%e(%l): %k %d: %m
"`

produces a message in following format:

X:\C.C(3): information C2901: Unrolling loop

See also:

- “ERRORFILE: Error File Name Specification”
- “-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode”
- “-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode”
- “-WmsgFonp: Message Format for No Position Information”
- “-WmsgFoi: Message Format for Interactive Mode”
- “-WmsgFob: Message Format for Batch Mode”

-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 no file information is available for a message (e.g. if a message is not related to a specific file). Then this message format string is used. Following formats are supported:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%K</td>
<td>Uppercase kind</td>
<td>ERROR</td>
</tr>
<tr>
<td>%k</td>
<td>Lowercase kind</td>
<td>error</td>
</tr>
<tr>
<td>%d</td>
<td>Number</td>
<td>C1815</td>
</tr>
<tr>
<td>%m</td>
<td>Message</td>
<td>text</td>
</tr>
<tr>
<td>%%</td>
<td>Percent</td>
<td>%</td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
<td></td>
</tr>
<tr>
<td>&quot;%&quot;</td>
<td>A &quot; if the filename, if the path or the extension contains a space</td>
<td></td>
</tr>
<tr>
<td>'%&quot;</td>
<td>A ' if the filename, the path or the extension contains a space</td>
<td></td>
</tr>
</tbody>
</table>

**Example:**
```
-\WmsgFonp"%k %d: %m\n"
```
produces a message in following format:
```
information L10324: Linking successful
```

**-WmsgFonp: Message Format for No Position Information**

**Group:**
MESSAGE

**Syntax:**
```
"-\WmsgFonp"<string>.
```

**Arguments:**

<string>: format string.

**Default:**
```
-\WmsgFonp"%f%e%": %K %d: %m\n"
```
Libmaker Options

Options

Description:
Sometimes no position information is available for a message (e.g. if a message is not related to a certain position). Then this message format string is used. Following formats are supported:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%K</td>
<td>Uppercase kind</td>
<td>ERROR</td>
</tr>
<tr>
<td>%k</td>
<td>Lowercase kind</td>
<td>error</td>
</tr>
<tr>
<td>%d</td>
<td>Number</td>
<td>C1815</td>
</tr>
<tr>
<td>%m</td>
<td>Message</td>
<td>text</td>
</tr>
<tr>
<td>%</td>
<td>Percent</td>
<td>%</td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
<td></td>
</tr>
<tr>
<td>%*</td>
<td>A &quot; if the filename,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if the path or the extension contains a space</td>
<td></td>
</tr>
<tr>
<td>%'</td>
<td>A ' if the filename,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>the path or the extension contains a space</td>
<td></td>
</tr>
</tbody>
</table>

Example:

-WmsgFonf"%k %d: %m\n"

produces a message in following format:

information L10324: Linking successful

-WmsgNe: Number of Error Messages

Group:
MESSAGE

Syntax:

"-WmsgNe" <number>.

Arguments:

<number>: Maximum number of error messages.
Libmaker Options
Options

Default:
50

Description:
Use this option to set the number of error messages that can occur before the Compiler stops.

NOTE Subsequent error messages caused by a previous message may not be directly related.

Example:
-WmsgNe2
Compiler stops after two error messages.

See also:
- “-WmsgNi: Number of Information Messages”
- “-WmsgNw: Number of Warning Messages”

-WmsgNi: Number of Information Messages

Group:
MESSAGE

Syntax:
“-WmsgNi" <number>.

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

Default:
50

Description:
Use this option to set the number of information messages to be logged.
**Example:**

- `WmsgN10`
  Only ten information messages are logged.

**See also:**

- `-WmsgNe: Number of Error Messages`
- `-WmsgNw: Number of Warning Messages`

**-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 that are not in the normal message categories (WARNING, INFORMATION, ERROR, FATAL). Use this option to disable such messages. This option can be used to reduce the number of messages and simplify error parsing of other tools.

- `a`: This argument disables messages relating to include files.
- `b`: This argument disables messages relating to files read, e.g. input files.
- `c`: This argument disables messages relating to generated files.
Libmaker Options

Options

d : This argument disables messages relating to statistics, e.g. code size and RAM/ROM usage.

e : This argument disables messages relating to informal messages (e.g. memory model, floating point format, etc.).

NOTE  Depending on the application, not all arguments may be applicable. In this case they are ignored.

Example:

-WmsgNu=c

-WmsgNw: Number of Warning Messages

Group:

MESSAGE

Syntax:

"-WmsgNw" <number>.

Arguments:

<number>: Maximum number of warning messages.

Default:

50

Description:

Use this option to set the number of warning messages.

Example:

-WmsgNw15

Only 15 warning messages are logged.

See also:

• “-WmsgNe: Number of Error Messages”
• “-WmsgNi: Number of Information Messages”
-WmsgSd: Disabling a Message

Group:
MESSAGE

Syntax:
"-WmsgSd" <number>.

Arguments:
<number>: Message number to be disabled, e.g. 1801

Default:
None.

Description:
Use this option to disable a message, so it does not appear in the error output.

Example:
-WmsgSd1801 disables the message for “implicit parameter declaration”

See also:
- “-WmsgSi: Set Message Type to Information”
- “-WmsgSw: Setting Message Type to Warning”
- “-WmsgSe: Setting Message Type to Error”

-WmsgSe: Setting Message Type to Error

Group:
MESSAGE

Syntax:
"-WmsgSe" <number>.
Libmaker Options

Arguments:

<number>: Message to be set as an error, e.g. 1853

Default:
None.

Description:
Allows a message to be redefined as an error message.

Example:

COMPOTIONS=-WmsgSe1853

See also:

- "-WmsgSd: Disabling a Message"
- "-WmsgSi: Set Message Type to Information"
- "-WmsgSw: Setting Message Type to Warning"

-WmsgSi: Set Message Type to Information

Group:
MESSAGE

Syntax:

"-WmsgSi" <number>.

Arguments:

<number>: Message to be redefined, e.g. 1853

Default:
None.

Description:
Use this option to redefine a message as an information message.

Example:

-WmsgSi1853
See also:

- “-WmsgSd: Disabling a Message”
- “-WmsgSi: Set Message Type to Information”
- “-WmsgSw: Setting Message Type to Warning”
- “-WmsgSe: Setting Message Type to Error”

-WmsgSw: Setting Message Type to Warning

Group:
MESSAGE

Syntax:

“-WmsgSw” <number>.

Arguments:

<number>: Message to be redefined, e.g. 2901

Default:

None.

Description:

Use this option to redefine a message to a warning message.

Example:

-WmsgSw2901

See also:

- “-WmsgSd: Disabling a Message”
- “-WmsgSi: Set Message Type to Information”
- “-WmsgSe: Setting Message Type to Error”
-WOutFile: Create Error List File

**Group:**

MESSAGE

**Syntax:**

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

**Arguments:**

None.

**Default:**

Error list file is created.

**Description:**

This option controls whether or not an error log file is created. The error file contains a list of all messages and errors created during processing. Since text error feedback can be handled with pipes to the calling application, it is possible to obtain this feedback without an explicit file. The name of the file is controlled by the environment variable "ERRORFILE: Error File Name Specification".

**Example:**

- WOutFileOn
  The error file is created, as specified with the ERRORFILE environment variable.

- WOutFileOff
  No error file is created.

**See also:**

- "-WErrFile: Create “err.log” Error File"
- "-WStdout: Write to Standard Output"
Libmaker Options
Options

-WStdout: Write to Standard Output

Group:
MESSAGE

Syntax:
"-WStdout" {"On" | "Off"}.

Arguments:
None.

Default:
Output is written to stdout

Description:
With Windows applications, the standard streams are available; but text written into them
does not appear anywhere unless explicitly requested by the calling application. With this
option text can be written to stdout.

Example:

-WStdoutOn
All messages are written to stdout.

-WErrFileOff
Nothing is written to stdout.

See also:
• “-WErrFile: Create “err.log” Error File”
• “-WOutFile: Create Error List File”
Libmaker Messages

This chapter describes messages produced by the Application.

Message Types

There are five types of messages generated:

**INFORMATION**
A message is displayed and compiling continues. Information messages indicate actions taken by the application.

**WARNING**
A message is displayed and processing continues. Warning messages indicate possible programming errors.

**ERROR**
A message is displayed and processing stops. Error messages indicate illegal use of the language.

**FATAL**
A message is displayed and processing is aborted. A fatal message indicates a severe error that will stop processing.

**DISABLE**
The message has been disabled. No message will be issued and processing will continue. The application ignores disabled messages.
Message Details

If the application displays a message, the message contains a message code and four to five digit number. This number can be used to search for the message in the help file. Following message codes are supported:

- “A” for Assemblers
- “B” for Burner
- “C” for Compilers
- “D” for Decoder
- “L” for Linker
- “LM” for Libmaker
- “M” for Maker

All messages generated by the application are documented in increasing order. Each message has a description and, if available, a brief example with possible solution or tips to fix the problem.

For each message, the type of message is also noted, e.g. [ERROR] indicates that the message is an error message.

[DISABLE, INFORMATION, WARNING, ERROR] indicates that the message is a warning message by default, but the user can change the message to DISABLE, INFORMATION or ERROR.

Message List

The following pages describe all messages documented at the time of this release.

LM1: Unknown Message Occurred

Message Type

[FATAL]

Description

The application tried to emit a message that was not defined. This is an internal error that should not occur. Please report any occurrences to your distributor.
LM2: Message Overflow, Skipping <kind> Messages

Message Type
[INFORMATION]

Description
The application displayed the number of messages of the specific type, as specified by the options:
- "-WmsgNi: Number of Information Messages",
- "-WmsgNw: Number of Warning Messages”, and
- "-WmsgNe: Number of Error Messages”.
Additional messages of this type that exceed the specified limit will not be displayed.

TIP
Use the options listed above to specify the number of messages that can be displayed.

LM50: Input File ‘<file>’ Not Found

Message Type
[FATAL]

Description
The Application was not able to find a file needed for processing.

TIP
Check whether the file really exits. Check if you are using a file name containing spaces (in this case you have to put quotes around it).

LM51: Cannot Open Statistic Log File <file>

Message Type
[WARNING]
Libmaker Messages

Message List

Description
It was not possible to open a statistic output file, therefore no statistics are generated.

NOTE  Not all tools support statistic log files. Even if a tool does not support it, the message still exists, but is never issued in this case.

LM52: Error in Command Line <cmd>

Description
In case there is an error while processing the command line, this message is issued.

LM64: Line Continuation Occurred in <FileName>

Description
In any environment file, the character '\\' at the end of a line is interpreted as a line continuation character. Because the path separation character for MS-DOS is also '\\', paths are often incorrectly written if they end with '\\'. Instead use a '.' after the last '\' to distinguish a path from a line continuation character.

Example

Current Default.env:

  ...
  LIBPATH=c:\freescale\lib
  OBJPATH=c:\freescale\work
  ...


Libmaker Messages
Message List

Is interpreted as
...
LIBPATH=c:\freescale\lib
OBJPATH=c:\freescale\work
...
To fix it, append a '.' after the '\'...
LIBPATH=c:\freescale\lib\.
OBJPATH=c:\freescale\work
...

NOTE Because this information occurs during the initialization phase of the application, the message prefix might not occur in the error message. So it might occur as “64: Line Continuation occurred in <FileName>”.

LM65: Environment Macro Expansion Message '<description>' for <variablename>

Message Type
[ERROR]

Description
During an environment variable macro substitution a problem occurred. Possible causes could be 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}
...

TIP Check the definition of the environment variable.
Libmaker Messages
Message List

LM66: Search Path <Name> Does Not Exist

Message Type
  [INFORMATION]

Description
The tool searched for a file that was not found. During the failed search for the file, a non-existing path was encountered.

TIP
Check the spelling of your paths. Update the paths when moving a project. Use relative paths.
Environment Variables

This chapter contains a short summary of various environment variables used by other utilities that you may encounter while using the Libmaker utility.

Directories

Source Files, Linker Parameter File

Source Files and the Linker Parameter File are searched first in the current directory, then in the other directories defined by the environment variable GENPATH.

Header Files

If a header file is included in double quotes, the current directory is searched first, then the directories given in GENPATH and finally those given in LIBPATH.

If it is included using angle brackets, the directories in GENPATH are not searched, only the current directory and those specified in LIBPATH.

Symbol Files

The compiler looks for symbol files in the current directory, then in the directories given by the environment variable SYMPATH and finally in directories given in GENPATH.

New symbol files are written in the directory containing the source, unless the environment variable SYMPATH is set. If set, the compiler puts the symbol file in the first directory in the path list.

Object Files

The linker and debugger look for object files in the current directory, then in directories specified in the environment variable OBJPATH and finally in GENPATH.

The compiler normally puts object files in the first directory specified in the environment variable OBJPATH. If that variable is not set, the object file is written into the directory containing the source file.
Absolute Files

The debugger looks for absolute files in the current directory, then in directories specified in ABSPATH and finally in GENPATH.

The linker creates absolute files in the first directory specified in ABSPATH. If that variable is not set, the absolute file is generated in the directory containing the parameter file.

Map Files

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 specified by the environment variable TEXTPATH.

Other Environment Variables

This section describes all other environment variables that may be set in the system.

Compiler Variables

COMPOPTIONS: If this variable is set, the compiler appends its contents to its command line each time a file is compiled. It can be used to globally specify certain options that should always be set, so you don’t have to specify them at each compilation.

Make Utility Variables

The make utility can access any environment variable with the following syntax: $\text{Name}$, e.g. $\text{COMP}$. For make files given in your installation, the following environment variables are used.

- **COMP**: contains name of Compiler
- **LINK**: contains name of Linker
- **FLAGS**: contains command line options for the compiler specified by COMP.

ERRORFILE: Error File Name Specification

The environment variable ERRORFILE specifies the name of the error file (used by the compiler or assembler).

Possible format specifiers are:

- ‘%n’: Substitute with the file name, without path.
- ‘%p’: Substitute with the path of source file.
Environment Variables
Directories

- `%f`: Substitute with 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 appears.

**Examples**

ERRORFILE=MyErrors.err
- lists all errors in the file MyErrors.err in the current directory.

ERRORFILE=\tmp\errors
- lists all errors in the file errors in the directory \tmp.

ERRORFILE=%f.err
- lists all errors in a file with the same name as the source file, but with extension
  .err, in the same directory as the source file. For example, if we compile a file
  \sources\test.c, an error list file \sources\test.err will be generated.

ERRORFILE=\dir1\%n.err
- For a source file test.c, an error list file \dir1\test.err will be generated.

ERRORFILE=%p\errors.txt
- For a source file \dir1\dir2\test.c, an error file
  \dir1\dir2\errors.txt will be generated.

**WARNING!** An existing file with the same name will be overwritten.

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

Directories
EBNF Notation

This chapter gives a brief overview of the Extended Backus–Naur Form (EBNF) notation, which is frequently used in this manual to describe file formats and syntax rules.

Introduction to EBNF

EBNF is frequently used in this reference manual to describe file formats and syntax rules. Therefore a short introduction to EBNF is given here.

**EBNF Example:**

\[
\text{ProcDecl} \quad = \quad \text{PROCEDURE} \quad "(" \text{ArgList } "\". \\
\text{ArgList} \quad = \quad \text{Expression} \quad (\"","\" \text{Expression}) \quad . \\
\text{Expression} \quad = \quad \text{Term} \quad (\"","\" \text{Term}) \quad . \\
\text{Term} \quad = \quad \text{Factor} \quad \text{AddOp} \quad \text{Factor} \quad . \\
\text{AddOp} \quad = \quad \"+\"|\"-\". \\
\text{Factor} \quad = \quad ([\"-\"] \text{Number})|(" \text{Expression } ")\quad .
\]

The EBNF language is a formalism that can be used to express the syntax of context-free languages. An EBNF grammar is a set of rules called *productions* of the form:

**LeftHandSide** = **RightHandSide**.

The left hand side is a so-called non-terminal symbol, the right hand side describes how it is composed.

EBNF consists of the following symbols:

- **Terminal symbols** (terminals for short) are the basic symbols which form the language described. In above example, the word **PROCEDURE** is a terminal. Punctuation symbols of the language described (not of EBNF itself) are quoted (they are terminals, too), while other terminal symbols are printed in **boldface**.

- **Non-terminal symbols** (non-terminals) are syntactic variables and have to be defined in a production, i.e. they have to appear on the left hand side of a production somewhere. In above example, there are many non-terminals, e.g. **ArgList** or **AddOp**.

- The vertical bar * * denotes an alternative, i.e. either the left or the right side of the bar can appear in the language described, but one of them has to. E.g. the 3rd
EBNF Notation

Introduction to EBNF

production above means "an expression is a term followed by either a "*" or a "/" followed by another term".

Parts of an EBNF production enclosed by "[ " and "] " are optional. They may appear exactly once in the language, or they may be skipped. The minus sign in the last production above is optional, both –7 and 7 are allowed.

- The repetition is another useful construct. Any part of a production enclosed by "{ " and "} " may appear any number of times in the language described (including zero, i.e. it may also be skipped). ArgList above is an example: an argument list is a single expression or a list of any number of expressions separated by commas. (Note that the syntax in the example does not allow empty argument lists...)

- For better readability, normal parentheses may be used for grouping EBNF expressions, as is done in the last production of the example. Note the difference between the first and the second left bracket: the first one is part of EBNF itself, the second one is a terminal symbol (it is quoted) and therefore may appear in the language described.

- A production is always terminated by a period.

EBNF-Syntax

We can now give the definition of EBNF in EBNF itself:

Production = NonTerminal "=" Expression ".".

Expression = Term {"|" Term}.

Term = Factor {Factor}.

Factor = NonTerminal
        | Terminal
        | "(" Expression ")" *
        | "[" Expression "]" *
        | "{" Expression "}" *.

Terminal = Identifier | "*** <any char> ***".

NonTerminal = Identifier.

The identifier for a non-terminal can be any name you like, terminal symbols are either identifiers appearing in the language described or any character sequence that is quoted.
Extensions

In addition to this standard definition of EBNF, we use the following notational conventions:

- The counting repetition: Anything enclosed by "{" and "}" and followed by a superscripted expression \( x \) must appear exactly \( x \) times. \( x \) may also be a non-terminal. In the following example, exactly four stars are allowed:

  \[
  \text{Stars} = \{**\}^4.
  \]

- The size in bytes. Any identifier immediately followed by a number \( n \) in square brackets ("[" and "]") may be assumed to be a binary number with the most significant byte stored first, having exactly \( n \) bytes. Example:

  \[
  \text{Struct} = \text{RefNo FilePos[4]}.
  \]

- In some examples, we enclose text by "<" and ">". This text is a meta–literal, i.e. whatever the text says may be inserted in place of the text. (cf. \(<\text{any char}>\) in the above example, where any character can be inserted).
Decoder

Introduction

This section describes the CodeWarrior IDE ELF/Freescale Decoder utility, which disassembles object files, absolute files and libraries in the Freescale (former Hiware) object file format or ELF/DWARF format and S-Record files. Various output formats are available.

The chapters in this section are:

- Decoder Controls: Pull-down menus and the Graphical User Interface (GUI)
- “Decoder Environment”: Describes the environment variables used by the Decoder
- “Input and Output Files”: Describes Decoder input and output files
- “Decoder Options”: Options that you can use to control Decoder operation
- “Decoder Messages”: Messages produced by the application

Product Highlights

The decoder utility has:

- Graphical User Interface (GUI)
- On-line Help
- Message Management
- 32-bit Functionality
- Decodes Freescale (formerly Hiware) object file format
- Decodes ELF/DWARF 1.1 and 2.0 object file format
- Decoder S-Record files
User Interface

The decoder provides a command line interface and an interactive interface (GUI). If no arguments are given on the command line, a window is opened that prompts for arguments.

The Decoder accepts object or absolute files, libraries, and S-Record files as input to generate the listing file. The name of the source files are encoded in the object or absolute file or library. For S-Record files, the processor must be specified with the option `-Env: Set Environment Variable`.

The generated listing file has the same name as the input file but with extension `.LST`. It contains source and assembly statements. The corresponding C/C++ source statements can be displayed within the generated assembly instructions.
Decoder Environment

This chapter describes the environment variables used by the Decoder. Some of them may be used by other tools such as the Macro Assembler or the Compiler.

Click any of the following links to jump to the corresponding section of this chapter:

- Settings
- Paths
- Line Continuation
- Environment Variables

Settings

Various settings for the Decoder may be set in an environment using environment variables. The syntax is always the same and given below:

Parameter = KeyName = ParamDef.

**NOTE**  No blank spaces are allowed in the definition of an environment variable.

**Example:**

```
GENPATH=C:\INSTALL\LIB;D:\PROJECTS\TESTS;\usr\local\lib;
```

These parameters may be defined in several ways:

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

**NOTE**  The maximum length of environment variable entries in the DEFAULT.ENV/.hidefaults is 1024 characters.
When looking for an environment variable, all programs first search the system environment, then the current project file (usually project.ini), then the DEFAULT.ENV file. If no definition can be found, a default value is assumed.

**NOTE** You can set the project directory via the DEFAULTDIR system environment variable. The environment may also be changed using the **Env: Set Environment Variable** Decoder option.

### Paths

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

\[\text{PathList} = \text{DirSpec }|;| \text{ DirSpec}\].

Example:

\[\text{GENPATH}=\text{C:\INSTALL\LIB;}\text{D:\PROJECT\TEST;}\text{usr\loc\freescale\lib;}\text{home\me}\]

If a directory name is preceded with an asterisk, programs recursively search in the given directory and subdirectories. Directories are parsed in the order they appear in the path list.

Example:

\[\text{LIBPATH}=\text{*C:\INSTALL\LIB}\]

**NOTE** Some DOS/UNIX environment variables (like GENPATH, LIBPATH, etc.) are used. For more detail refer to “Environment” section.

Environment variables are stored usually in the project file (for example, project.pjt) and can be modified in the IDF. Adapt the project properties where the environment can be modified.

**NOTE** When using an external editor (WinEdit, Codewright), do not set the system environment variable DEFAULTDIR. If you do so and this variable does not contain the project directory given in the editor’s project configuration, files might not be located where you expect them to be saved.
Line Continuation

It is possible to specify an environment variable in an environment file (project.pjt or .hidefaults) over multiple lines with the line continuation character '\':

**Example:**

```plaintext
COMPOPTIONS=\n-W2 \n-Wpd
```

This is the same as

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

But this feature is interpreted differently when used with paths, for example:

```plaintext
GENPATH=.\nTEXTFILE=.\txt
```

will be interpreted as

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

To avoid such problems, use a semicolon ';' at the end of a path if there is a '\' at the end, as shown below:

```plaintext
GENPATH=.\;
TEXTFILE=.\txt
```
Environment Variables

This section describes each environment variable available for the Decoder. Variables are listed in alphabetical order. Topics describing each variable’s details are in Table 25.1.

Table 25.1  Environment Variables

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tools</td>
<td>For some environment variables, a synonym also exists. Those synonyms may be used for earlier releases of the Decoder and will be removed in the future. A synonym has lower precedence than the environment variable.</td>
</tr>
<tr>
<td>Synonym</td>
<td>Specifies the syntax of the option in EBNF format.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Describes and lists optional and required arguments for the variable.</td>
</tr>
<tr>
<td>Arguments</td>
<td>Shows the default setting for the variable or none.</td>
</tr>
<tr>
<td>Description</td>
<td>Provides a detailed description of the option and how to use it.</td>
</tr>
<tr>
<td>Example</td>
<td>Gives an example of usage, and effects of the variable when possible. Shows an entry in the default.env for PC or in the .hidefaults for UNIX.</td>
</tr>
</tbody>
</table>

**DEFAULTDIR: Current Directory**

**Tools:**

Compiler, Assembler, Linker, Decoder, Debugger, Libmaker, Maker

**Synonym:**

None

**Syntax:**

```
DEFAULTDIR= <directory>.
```

**Arguments:**

`<directory>`: Default current directory
Decoder Environment

Environment Variables

Default:

None

Description:

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

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

Example:

DEFAULTDIR=C:\INSTALL\PROJECT

ENVIRONMENT: Environment File Specification

Tools:

Compiler, Linker, Decoder, Debugger, Libmaker, Maker

Synonym:

HIENVIRONMENT

Syntax:

ENVIRONMENT= <file>.

Arguments:

<file>: file name with path specification

Default:

DEFAULT.ENV on PC, .hidefaults on UNIX

Description:

You must specify this variable at the system level. Usually the Decoder looks in the current directory for an environment file named DEFAULT.ENV (.hidefaults on UNIX). Using ENVIRONMENT (set in AUTOEXEC.BAT (DOS) or .cshrc (UNIX)), a different file name may be specified.
Decoder Environment

Environment Variables

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

Example:

ENVIRONMENT=FREESCALE\prog\global.env

GENPATH: Defines Paths to Search for Input Files

Tools:
Compiler, Assembler, Linker, Decoder, Debugger

Synonym:
HIPATH

Syntax:

GENPATH= {<path>}

Arguments:

<path>: Paths separated by semicolons

Description:
The Decoder looks for the required input files (binary input files and source files) first in
the project directory, then in the directories listed in the environment variable GENPATH.

NOTE If a directory specification in this environment variable starts with an asterisk,
the whole directory tree is searched recursively. All subdirectories and their
subdirectories are searched. Within one level in the tree, search order of the
subdirectories is indeterminate.

Example:

GENPATH=\obj;..\..\lib;
TEXTPATH: Text Path

Tools:
  Compiler, Assembler, Linker, Decoder

Synonym:
  None

Syntax:
  TEXTPATH= {<path>}.

Arguments:
  <path>: Paths separated by semicolons

Description:
When you define this environment variable, the Decoder stores the list file it produces in the first directory specified. If TEXTPATH is not set, the generated .LST file is stored in the directory in which the binary input file was found.

Example:
  TEXTPATH=\sources ..\..\headers;\usr\local\txt
Input and Output Files

This chapter describes Decoder input and output files.

- **Input Files**
- **Output Files**

### Input Files

Input files include the following file types:

- Absolute files
- Object files
- S-Record files
- Intel Hex files

### Absolute Files

The decoder takes any file as input, it does not require the file name to have a special extension. However, we suggest that all your absolute file names have extension `.ABS`. Absolute files will be searched first in the project directory and then in the directories listed in `GENPATH`. The absolute file must be a valid ELF/DWARF V1.1, ELF/DWARF V2.0, or Freescale absolute file.

**NOTE** For Freescale (former Hiware) absolute files, no source information is decoded because the absolute file does not contain source information.

### Object File

The decoder takes any file as input, it does not require the file name to have a special extension. However, we suggest that all your relocatable file names have extension `.O`. Object files will be searched first in the project directory and then in the directories listed in `GENPATH`. The object file must be a valid ELF/DWARF V1.1, ELF/DWARF V2.0, or Freescale (former Hiware) relocatable file.
Input and Output Files

Output Files

S-Record Files
For S-Record files, the processor must be specified with the option `-Proc: Set Processor`. Otherwise only the structure of the S-Record file is printed, but the code is not disassembled.

Intel Hex Files
For Intel Hex files the same applies as for S-Record Files. To disassemble the code, the processor must be specified with the option `-Proc: Set Processor`.

Output Files
After a successful decoding session, the Decoder generates a listing file containing the disassembled instructions generated by each source statement. This file is written to the directory given in the environment variable `TEXTPATH`. If that variable contains more than one path, the listing file is written in the first directory given. If this variable is not set, the listing file is written in the directory containing the binary input file. Listing files always get the extension `.LST`.

In a standard listing file, the code depends on the target. A sample listing is as follows:

```
DISASSEMBLY OF: '.text' FROM 331 TO 416 SIZE 85 (0X55)
Source file: 'Y:\DEMO\WAVE12C\fibo.c'
8: unsigned int Fibonacci(unsigned int n)
Fibonacci:
00000867 1B98         LEAS  -8,SP
00000869 3B           PSHD
13:   fib1 = 0;
0000086A C7           CLRB
0000086B 87           CLRA
0000086C 6C88         STD   8,SP
14:   fib2 = 1;
0000086E 52           INCB
0000086F 6C84         STD   4,SP
15:   fibo = n;
00000871 EE80         LDX   0,SP
00000873 6E86         STX   6,SP
16:   i = 2;
00000875 58           ASLB
00000876 6C82         STD   2,SP
17:   while (i <= n) {
00000878 2011         BRA   *+19   ;abs = 088B
18:   fibo = fib1 + fib2;
```
0000087A EC88     LDD    8,SP
0000087C E384     ADDD   4,SP
0000087E 6C86     STD    6,SP
  19:     fib1 = fib2;
00000880 EE84     LDX    4,SP
00000882 6E88     STX    8,SP
  20:     fib2 = fibo;
00000884 6C84     STD    4,SP
  21:     i++;
00000886 EE82     LDX    2,SP
00000888 08       INX
00000889 6E82     STX    2,SP
  17:     while (i <= n) {
0000088B EC82     LDD    2,SP
0000088D AC80     CPD    0,SP
0000088F 23E9     BLS   *-21 ; abs = 087A
  23:     return(fibo);
00000891 EC86     LDD    6,SP
  24:   }
00000893 1B8A     LEAS   10,SP
00000895 3D       RTS
Input and Output Files

Output Files
Decoder Options

The Decoder offers options that you can use to control its operation. Options are composed of a dash (or minus sign) followed by one or more letters or digits. You can specify decoder options on the command line. Command line options are not case sensitive; for example, `–otest.lst` is the same as `–OTEST.LST`.

**NOTE** Arguments for an option must not exceed 128 characters.

Anything not starting with a dash is the name of a parameter file to be linked. Options for the Freescale (former Hiware) object file format may differ from the options for decoding ELF/DWARF binaries.

Using Decoder Options

This section lists and describes each Decoder option.

Option Topics

Table 27.1 describes topics (details) in each option listing.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td>Specifies one of four option groups:</td>
</tr>
<tr>
<td></td>
<td>- OUTPUT: Format and content of the listing file.</td>
</tr>
<tr>
<td></td>
<td>- INPUT: Control and specification of input files</td>
</tr>
<tr>
<td></td>
<td>- HOST: Host and Operation System-dependent options</td>
</tr>
<tr>
<td></td>
<td>- MESSAGE: Control of error and message output</td>
</tr>
<tr>
<td>Syntax</td>
<td>Specifies the syntax of the option in EBNF format.</td>
</tr>
<tr>
<td>Arguments</td>
<td>Describes and lists optional and required arguments for the option.</td>
</tr>
<tr>
<td>Default</td>
<td>Shows the option's default setting</td>
</tr>
<tr>
<td>Description</td>
<td>Provides a detailed description of the option and how to use it</td>
</tr>
</tbody>
</table>
**Special Modifiers**

You can use special modifiers with some options, although some modifiers may not make sense for all options. Table 27.2 lists and describes these modifiers.

### Table 27.2 Supported Modifiers

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%p</td>
<td>path including file separator</td>
</tr>
<tr>
<td>%N</td>
<td>file name in strict 8.3 format</td>
</tr>
<tr>
<td>%n</td>
<td>file name without extension</td>
</tr>
<tr>
<td>%E</td>
<td>extension in strict 8.3 format</td>
</tr>
<tr>
<td>%e</td>
<td>extension</td>
</tr>
<tr>
<td>%f</td>
<td>path + file name without extension</td>
</tr>
<tr>
<td>%&quot;</td>
<td>a double quote (&quot;) if the file name, path or extension contains a space</td>
</tr>
<tr>
<td>%'</td>
<td>a single quote (’) if the file name, path or extension contains a space</td>
</tr>
<tr>
<td>%(ENV)</td>
<td>replaces it with contents of an environment variable</td>
</tr>
<tr>
<td>%%</td>
<td>generates a single '%'</td>
</tr>
</tbody>
</table>

### Examples

For these examples we assume that our actual file name (base file name for the modifiers) is:

```plaintext
c:\Freescale\my demo\TheWholeThing.myExt
```

%p gives the path only with a file separator:

```plaintext
c:\Freescale\my demo\%
```

%N results in the file name in 8.3 format, that is the name with only 8 characters:

```plaintext
TheWhole
```

%n returns just the file name without extension:

```plaintext
TheWholeThing
```

%E gives the extension in 8.3 format, that is the extension with only 3 characters:

```plaintext
myE
```
Decoder Options
Using Decoder Options

%e is used for the whole extension:
myExt
%f gives the path plus the file name:
c:\Freescale\my demo\TheWholeThing
Because the path contains a space, using %" or %’ is recommended: Thus %”%f%” gives:
c:\Freescale\my demo\TheWholeThing
where %’%f’%’ gives:
‘c:\Freescale\my demo\TheWholeThing’
When using % (envVariable) an environment variable may be used too. A file separator after % (envVariable) is ignored if the environment variable is empty or does not exist. For example, $(TEXTPATH)\myfile.txt is replaced with:
c:\Freescale\txt\myfile.txt
if TEXTPATH is set to:
TEXTPATH=c:\Freescale\txt
But is set to:
myfile.txt
if TEXTPATH does not exist or is empty.
%% may be used to print a percent sign. %e%% gives:
myExt%

-A: Print Full Listing

Group:
OUTPUT
Syntax:
-A
Arguments:
None
File Format:
Only Freescale (former Hiware). (ELF Object files are not affected by this option.)
Decoder Options
Using Decoder Options

**Description:**
Print a listing with the header information of the object file.

**Example:**

Listing with command line fibo.o -A:

*** Header information ***
Program Version 2700
Format Version 2
File Id 129
flags 0
processor family 11
processor type 1
Unitname fibo.abs
Username PFR
Program time string Feb 25 1998
CopyRight
*** Directory information for Absfile***
Is romlib? 0
Init start:end 32774:32774
Code beg:end 32768:32939
Data beg:end 384:4096
Total number of objects 7
At address: 8000 code size: 40
00008000 1410 ORCC #16
...........

---

**-C: Write Disassembly Listing With Source Code**

**Group:**
OUTPUT

**Syntax:**

-C

**Arguments:**

None
Decoder Options

Using Decoder Options

Default:
None

File Format:
Only Freescale (former Hiware). (ELF Object files are not affected by this option.)

Description:
This option setting is default for the Freescale (former Hiware) object files as input. When this option is specified, the Decoder decoding Freescale object files writes the source code within the disassembly listing.

Example:
Listing with command line fibo.o -C (code depends on target):
8:  unsigned int Fibonacci(unsigned int n)
9:  {
10:   unsigned fib1, fib2, fibo;
11:   int i;
12:
00000000 3B           PSHD
00000001 3B           PSHD
13:   fib1 = 0;
00000002 C7           CLRB
00000003 87           CLRA
14:   fib2 = 1;
00000004 52           INCB
00000005 6C82         STD   2,SP
15:   fibo = n;
00000007 EE80         LDX   0,SP
16:   i = 2;
 ....................

-D: Decode DWARF Sections

Group:
OUTPUT

Syntax:
-D
Decoder Options

Using Decoder Options

Arguments:
None

Default:
Disabled

File Format:
Only ELF. Freescale Object files are not affected by this option.

Description:
When you specify this option, DWARF section information is also written to the listing file. Decoding from the DWARF section inserts this information in the listing file. See the following listings for more information.

Listing 27.1 Source/code reference information

```
 данные
 0x4 Version 2
 0x6 PrologLen 1221
 0xa MinInstrLen 1c
 0xb DefIsStmt 0c
 0xc LineBase 0c
 0xd LineRange 4c
 0xe DW2L_OpcodeBase 9c
 0xf Opcodelengths : 0c 1c 1c 1c 1c 0c 0c 0c 1c
Includedir:
0x19 File 1: Y:\DEMO\WAVE12C\fibo.c, 0, 0, 0
0x33 File 2: y:\LIB\ELF12C\hidef.h, 0, 0, 0
0x4c File 3: y:\LIB\ELF12C\default.sgm, 0, 0, 0
0x69 File 4: y:\LIB\ELF12C\stddef.h, 0, 0, 0
0x84 Set Addr 867(2151): ADDR FILE LINE COL STMT BASIC
0x8b set column : 867 1 1 14 0 0
0x8d advance line : 867 1 8 14 0 0
0x8f negate stmt : 867 1 8 14 1 0
0x90 negate stmt : 867 1 8 14 0 0
...```

Listing 27.2 Argument location for local variables information

```
 данные
 .debug_loc
 0 Start 867, End 869 (2)DW_OP_breg15 0(0)
 0xc Start 869, End 86a (2)DW_OP_breg15 8(8)
 0x18 Start 86a, End 895 (2)DW_OP_breg15 10(a)
```
Decoder Options
Using Decoder Options

0x24 Start 895, End 896 (2)DW_OP_breg15 0(0)
0x30 0, 0 : end of location-list

Listing 27.3 Symbol Debug information

DWARF: .debug_info (1053) [0x734]
Compi.Unit Header: size 304, version 2, abbrev 0, addrsiz e 4
  0xb Abbrevation 128 ,compile_unit
  0xd name         string       fibo.c
  0x14 producer    string       FREESCALE
  0x1b comp_dir    string       Y:\DEMO\WAVE12C
  0x2b language    udata        DW_LANG_C89
  0x2c stmt_list   data4        0(0)

Listing 27.4 Frame Debug Information

.debug_frame
  0 CIE Information  0x8 Version 1
  0x9 Augmentor Freescale CFA 1.0
  0x18 CodeAlign: 1, DataAlign: 1, ReturnAddr-Column: 18
                      R[30] R[31]
  0x1b start-values                   84d:   0( 15)
  0x1b Def CFA Register reg: 15, 0x1d Def CFA Offset ofs: 0
  0x1f Offset: reg 18, Ofs: 0
  0x21 Undefined reg: 0
  0x23 Undefined reg: 1

NOTE Specify the –E option when the –D option is activated.
-E: Decode ELF sections

Group:
OUTPUT

Syntax:
-E

Arguments:
None

File Format:
Only ELF, Freescale (former Hiware) Object files are not affected by this option.

Description:
When you specify this option, ELF section information is also written to the listing file. Decoding from the ELF section inserts the following information in the listing file:

Listing 27.5  ELF Header Information

File: Y:\DEMO\WAVE12C\fibo.abs
Ident: ELF with 32-bit objects, MSB encoding, Version 1
Type: Executable file, Machine: Freescale HC12, Vers: 1
Entry point: 83D
Elf flags: 0
ElfHSiz: 34
ProgHOff: 34, ProgHSi: 20, ProgHNu: 6
SectHOff: E3A, SectHSi: 28, SectHNu: 19,
SectHSI: 18

Usually the ELF Program header Table is available only for absolute files.

Listing 27.6  ELF Program header Table Information

<table>
<thead>
<tr>
<th>NO</th>
<th>TYPE</th>
<th>OFFSET</th>
<th>SIZE</th>
<th>VIRTADDR</th>
<th>PHYADDR</th>
<th>MEMSIZE</th>
<th>FLAGS</th>
<th>ALIGNMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PT_PHDR</td>
<td>34</td>
<td>C0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>PT_LOAD</td>
<td>F4</td>
<td>0</td>
<td>800</td>
<td>4</td>
<td>6</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>PT_LOAD</td>
<td>F4</td>
<td>AE</td>
<td>810</td>
<td>AE</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Decoder Options
Using Decoder Options

Listing 27.7 ELF Section Header Table Information

SECTION HEADER TABLE - 19 Items
Starts at: E3A, Size of an entry: 28, Ends at: 1132
String table is in section: 12

<table>
<thead>
<tr>
<th>NO</th>
<th>NAME</th>
<th>TYPE</th>
<th>FLAGS</th>
<th>OFFSET</th>
<th>SIZE</th>
<th>ADDR</th>
<th>ALI</th>
<th>RECS</th>
<th>LINK</th>
<th>INFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-</td>
<td>NULL</td>
<td>NULL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1-.common</td>
<td>NOBITS</td>
<td>WA</td>
<td>F4</td>
<td>4</td>
<td>800</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2-.init</td>
<td>PROGBITS</td>
<td>AX</td>
<td>F4</td>
<td>3D</td>
<td>810</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3-.startData</td>
<td>PROGBITS</td>
<td>AX</td>
<td>131</td>
<td>1A</td>
<td>84D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4-.text</td>
<td>PROGBITS</td>
<td>AX</td>
<td>14B</td>
<td>55</td>
<td>867</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5-.copy</td>
<td>PROGBITS</td>
<td>AX</td>
<td>1A0</td>
<td>2</td>
<td>88C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6-.stack</td>
<td>NOBITS</td>
<td>WA</td>
<td>1A2</td>
<td>100</td>
<td>B00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7-.vectSeg0_vect</td>
<td>PROGBITS</td>
<td>AX</td>
<td>1A2</td>
<td>2</td>
<td>FFFE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Listing 27.8 Symbol Table Information

SYMBOL TABLE: .symtab - 13 Items
Starts at: 1A4, Size of an entry: 10, Ends at: 274
String table is in section: 9
First global symbol is in entry no.: 8

<table>
<thead>
<tr>
<th>NO</th>
<th>NAME</th>
<th>VALUE</th>
<th>SIZE</th>
<th>BIND</th>
<th>TYPE</th>
<th>SECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-</td>
<td></td>
<td>0</td>
<td>0</td>
<td>LOCAL</td>
<td>NOTYPE</td>
<td></td>
</tr>
<tr>
<td>1-</td>
<td></td>
<td>0</td>
<td>0</td>
<td>LOCAL</td>
<td>SECTION 1</td>
<td>1</td>
</tr>
<tr>
<td>2-</td>
<td></td>
<td>0</td>
<td>0</td>
<td>LOCAL</td>
<td>SECTION 2</td>
<td>2</td>
</tr>
<tr>
<td>3-Init</td>
<td></td>
<td>810</td>
<td>2D</td>
<td>LOCAL</td>
<td>FUNC</td>
<td>2</td>
</tr>
<tr>
<td>4-</td>
<td></td>
<td>0</td>
<td>0</td>
<td>LOCAL</td>
<td>SECTION 3</td>
<td>3</td>
</tr>
<tr>
<td>5-</td>
<td></td>
<td>0</td>
<td>0</td>
<td>LOCAL</td>
<td>SECTION 4</td>
<td>4</td>
</tr>
</tbody>
</table>

Listing 27.9 Relocation Section Information

RELOCATION TABLE RELA: .rela.init - 1 Items
Starts at: 2AA, Size of an entry: 10, C, Ends at: 2B6
Symbol table is in section: 8
Binary code/data is in section: 2

<table>
<thead>
<tr>
<th>NO</th>
<th>OFFSET</th>
<th>SYMNDX</th>
<th>TYP</th>
<th>ADDEND</th>
<th>SYMNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2163</td>
<td>873</td>
<td>3</td>
<td>3</td>
<td>4107</td>
</tr>
</tbody>
</table>

Listing 27.10 Hexadecimal dump from all sections defined in the binary file

HEXDUMP OF: .init FROM 244 TO 305 SIZE 61 (0X3D)
OFFSET  +0 +1 +2 +3 +4 +5 +6 +7 : +8 +9 +A +B +C +D +E +F ASCII DATA
000000  FE 08 55 PD 08 53 27 10 : 35 ED 31 EC 31 69 70 83 ...U ...
S".5.1.lip.
Decoder Options
Using Decoder Options

-Ed: Dump ELF Sections in LST File

Group:
OUTPUT

Syntax:
-Ed

Arguments:
None

Default:
None

File Format:
Only ELF. Freescale (former Hiware) object files are not affected by this option.

Description:
This option generates a HEX dump of all ELF sections.

NOTE The related option -E shows the information contained in ELF sections in a more readable form.
-Env: Set Environment Variable

**Group:**
HOST

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

**Arguments:**
- **<Environment Variable>:** Environment variable to be set
- **<Variable Setting>:** Environment variable value

**Default:**
None

**Description:**
This option sets an environment variable.

**Example:**
-Env OBJPATH=\sources\obj
  This is the same as:
  OBJPATH=\sources\obj
  in the default.env

-F: Object File Format

**Group:**
INPUT

**Syntax:**
-F ( A | E | I | H | S )

**Arguments:**
None
**Decoder Options**

*Using Decoder Options*

**Default:**

- `FA`

**Description:**

The decoder is able to decode different object file formats. This option defines which object file format should be decoded:

- `FA`: the decoder determines the object file format automatically.
- `FE`: this can be overridden and only ELF files are correctly decoded.
- `FH`: only Freescale (former Hiware) files are decoded.
- `FS`: only S-Record files can be decoded.
- `FI`: Intel Hex files can be decoded.

**NOTE** This option defines the Object File Format, which also defines the format of absolute files and libraries. It does not only affect object files. Many other options only effect a specific object file format. See the corresponding option for details.

**NOTE** To decode an S-Record or Intel Hex file, use the option `-Proc: Set Processor` to specify the processor.

- `-H`: Prints the List of All Available Options

**Group:**

OUTPUT

**Syntax:**

- `-H`

**Arguments:**

None

**Description:**

Prints the list of all Decoder options. The options are sorted by Group. Options in the same group are sorted alphabetically.
-L: Produce Inline Assembly File

**Group:**

OUTPUT.

**Syntax:**

- L

**Arguments:**

None

**File Format:**

Only Freescale (former Hiware). ELF Object files are not affected by this option.

**Description:**

The output listing is an inline assembly file without additional information, but in C comments.

**Example:**

Part of Listing with command line fibo.o -L (code depends on target):

```c
unsigned int Fibonacci(unsigned int n)
{
    unsigned fib1, fib2, fibo;
    int i;
    asm(
        CLRB
        CLRA
        INCB
        STD 2, SP
        LDX 0, SP
        LDY #2
        SEX A,D
        BRA LBL25
    LBL16:
        ADDD 2, SP
        ...
        RTS
    )
}
```
Decoder Options
Using Decoder Options

-Lic: Print License Information

Group:
Various

Syntax:
-Lic

Arguments:
None

Default:
None

Description:
This option shows the current state of the license information. When no full license is available, the Decoder runs in demo mode. Currently, there are no restrictions in the demo mode, but future versions might limit the size of the listing or the size of the input files.

Example:
None

-LicA: License Information About Every Feature in Directory

Group:
Various

Syntax:
-LicA

Arguments:
None
**-LicA: Option Prints License Information**

**Description:**
The `-LicA` option prints the license information of every tool or dll in the directory containing the executable. For example, if tool or feature is a demo version or full version. Because the option analyzes every file in the directory, this takes a long time.

**Example:**
None

**-LicBorrow: Borrow License Feature**

**Group:**
Host

**Syntax:**
-licborrow <feature>[ ; <version>] : <date>

**Arguments:**
- `<feature>`: the feature name to be borrowed (e.g. HI100100).
- `<version>`: optional version of the feature to be borrowed (e.g. 3.000).
- `<date>`: date with optional time until when the feature shall be borrowed (e.g. 15-Mar-2004:18:35).

**Default:**
None

**Description:**
This option allows to borrow a license feature until a given date/time. Borrowing allows you to use a floating license even if disconnected from the floating license server.

You need to specify the feature name and the date until you want to borrow the feature. If the feature you want to borrow is a feature belonging to the tool where you use this option, then you do not need to specify the version of the feature (because the tool knows the version). However, if you want to borrow any feature, you need to specify as well the feature version of it. You can check the status of currently borrowed features in the tool about box. You only can borrow features, if you have a floating license and if your floating license is enabled for borrowing. See as well the provided FLEXlm documentation about details on borrowing.
Example:

-LicBorrowHI100100;3.000:12-Mar-2004:18:25

-LicWait: Wait for Floating License from Floating License Server

Group:
Host

Syntax:

-LicWait

Arguments:
None

Description:
By default, if a license is not available from the floating license server, then the application will immediately return. With -LicWait set, the application will wait (blocking) until a license is available from the floating license server.

Example:
None

-N: Display Notify Box

Group:
Various

Syntax:

-N

Arguments:
None
**Decoder Options**

**Using Decoder Options**

---

**Description:**
Causes the Decoder to display an alert box if there was an error during decoding. This is useful when running a make file (refer to Make Utility), since the Decoder waits for the user to acknowledge the message, thus suspending make file processing. (The 'N' stands for “Notify”.)

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

**NOTE**
This option is only present on the PC version of the Decoder. The UNIX version does not accept `-n` as an option string.

**Example:**
None

---

**-NoBeep: No Beep in Case of an Error**

**Group:**
MESSAGE.

**Syntax:**
```
-NoBeep
```

**Arguments:**
None

**Description:**
Normally there is a beep at the end of processing if there was an error. This beep may be switched off with this option.

**Example:**
None

---

*HC(S)08/RS08 and HC(S)12 Build Tools Utilities Manual*
Decoder Options

Using Decoder Options

-NoEnv: Do not use Environment

Group:
Startup. This option cannot be specified interactively.

Syntax:
-NoEnv

Arguments:
None

Description:
This option can only be specified at the command line while starting the application. It can
not be specified in any other circumstance, including the default.env file or command line.
When this option is given, the application does not use any environment (default.env, project.ini or tips file).

Example:
    decoder.exe -NoEnv

-NoSym: No Symbols in Disassembled Listing

Group:
OUTPUT

Syntax:
-NoSym

Arguments:
None

Description:
No Symbols are printed in the disassembled listing.
NOTE In previous versions of the Decoder, this option was called -N. It was renamed because of a conflict with the common option -N, which was not present in previous versions. The option -NoSym has no effect when decoding .abs file. As the .abs file does not contain any relocation information, it is not possible to display symbol names in the disassembly listing.

Example:

Part of Listing with command line fibo.o -NoSym.

DISASSEMBLY OF: '.text' FROM 531 TO 664 SIZE 133 (0X85)
Source file: 'fibo.c'
   19: unsigned int Fibonacci(unsigned int n)
Fibonacci:  
00000000 1B98 LEAS -8,SP
00000002 3B PSHD
   24: fib1 = x[0] + f + g;
   00000003 FC0000 LDD $0000
   00000006 F30000 ADDD $0000
   00000009 F30000 ADDD $0000
   0000000C 6C88 STD 8,SP
   25: fib2 = x[1];
   0000000E FC0002 LDD $0002
   00000011 6C84 STD 4,SP

-O: Defines Listing File Name

Group:

OUTPUT

Syntax:

-O <FileName>

Arguments:

<fileName>: Name of listing file that must be generated by the decoding session.

Default:

None
**Decoder Options**

**Using Decoder Options**

---

**Description:**
This option defines the name of the output file to be generated.

**Example:**

```
-O=TEST.LST
```

The decoder generates a file named TEST.LST.

---

**-Proc: Set Processor**

**Group:**
INPUT

**Syntax:**

```
-Proc= <ProcessorName>[ : <Derivative>].
```

**Arguments:**

- `<ProcessorName>`: Name of a supported processor.
- `<DerivativeName>`: Name of supported derivative.

**Default:**
None

**Description:**
This option specifies which processor should be decoded. For object files, libraries and applications, the processor is usually detected automatically. For S-Record and Intel Hex files, however, the decoder cannot determine which CPU the code is for, and therefore the processor must be specified with this option to get a disassembly output. Without this option, only the structure of a S-Record file is decoded.

The following values are supported:

HC05, HC08, HC08:HC808, HC11, HC12, HC12:CPU12, HC12:HCS12, HC12:HCS12X, HC16, M68k, MCORE, PPC, RS08, 8500, 8300, 8051, ST7 and XA

**Example:**

```
decoder.exe fibo.s19 -proc=HC12
```
-T: Show Cycle Count for Each Instruction

**Group:**
OUTPUT

**Syntax:**
-T

**Arguments:**
None

**Description:**
If you specify this option, each instruction line contains the count of cycles in ‘[‘,’]’ braces. The cycle count is written before the mnemonics of the instruction. Note that the cycle count display is not supported for all architectures.

**Example:**
Part of Listing (HC12, ELF) with command line fibo.o -T:

```
DISASSEMBLY OF: '.text' FROM 531 TO 664 SIZE 133 (0X85)
Source file: 'X:\CHC12E\DEMO\ELF12C\fibo.c'
19: unsigned int Fibonacci(unsigned int n)
Fibonacci:
00000000 1B98         [2]    LEAS  -8,SP
00000002 3B           [2]    PSHD
24:   fib1 = x[0] + f + g;
00000003 FC0000       [3]    LDD   x
00000006 F30000       [3]    ADDD  f
00000009 F30000       [3]    ADDD  g
0000000C 6C88         [2]    STD   8,SP
25:   fib2 = x[1];
0000000E FC0002       [3]    LDD   x
00000011 6C84         [2]    STD   4,SP
26:   fibo = 0;
00000013 C7           [1]    CLRB
00000014 87           [1]    CLRA
00000015 6C86         [2]    STD   6,SP
27:   i = 2;
00000017 C602         [1]    LDAB  #2
00000019 6C82         [2]    STD   2,SP
```
**Decoder Options**

*Using Decoder Options*

---

**-V: Print Decoder Version**

**Group:**
- Various

**Syntax:**
- `-V`

**Arguments:**
- None

**Description:**
Prints the Decoder current directory and version information. Version of the complete Decoder is the main version. Additionally, version numbers for the Freescale (former Hiware) and ELF Object File Format decoding parts are printed separately.

---

**-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.
**Decoder Options**

**Using Decoder Options**

**Description:**

Normally an application is started as a normal window if no arguments are given. If the application is started with arguments (for example, from the maker to compile/link a file) then the application is running minimized to allow for 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 screen).

Using `-ViewHidden` the application processes arguments (files to be compiled/link) in the background (no window/icon in the taskbar visible). However, if you are using the “-N: Display Notify Box” option a dialog box is still possible.

**Example:**

```
-ViewMin fibo.o
```

**-WErrFile: Create “err.log” Error File**

**Group:**

MESSAGE

**Syntax:**

```
-WErrFile ( On | Off ).
```

**Arguments:**

None

**Default:**

```
err.log is created/deleted.
```

**Description:**

Error feedback from the compiler to called tools is now done with a return code. In 16-bit windows environments, this was not possible. An `err.log` file with the number of errors written was used to signal an error. To state no error, the `err.log` file was deleted. Using UNIX or WIN32, there is now a return code available, so this file is no longer needed when UNIX / WIN32 applications are involved. To use a 16-bit maker with this tool, the error file must be created in order to signal an error.
Decoder Options

Using Decoder Options

Example:

-\texttt{-WErrFileOn\ :}
  \texttt{err.log} is created/deleted when the application is finished.
-\texttt{-WErrFileOff\ :}
  existing \texttt{err.log} is not modified.

\texttt{-Wmsg8x3: Cut File Names in Microsoft Format to 8.3}

Group: MESSAGE

Syntax:

-\texttt{-Wmsg8x3}

Arguments:

None

Description:

Some editors (early versions of WinEdit) expect the file name in the Microsoft message format in strict 8.3 format. That means the file name can have at most eight characters with not more than a three character extension. With 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.

\textbf{NOTE}\quad This option is only present in PC versions of the Decoder. UNIX decoders do not accept \texttt{-Wmsg8x3}.

Example:

None
-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)

**Description:**

With this option it is possible to change the error message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and specified in decimal. Hexadecimal needs a 0X prefix, (for gray errors: -WmsgCE0x808080).

**Example:**

-WmsgCE255

Changes error messages to blue

---

- WmsgCF: RGB Color for Fatal Messages

**Group:**

MESSAGE

**Scope:**

Function
Decoder Options
Using Decoder Options

Syntax:
-WmsgCF <RGB>

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

Default:
-WmsgCF8388608 (r80 g00 b00, dark red)

Description:
With this option it is possible to change the fatal message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and specified in decimal. Hexadecimal needs a 0X prefix, (for gray errors: -WmsgCF0x808080).

Example:
-WmsgCF255
Changes the fatal messages to blue

-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)
**Decoder Options**

**Using Decoder Options**

---

**Description:**

With this option it is possible to change the information message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and specified in decimal. Hexadecimal needs a 0X prefix, (for gray errors: -WmsgCI0x808080).

**Example:**

- `-WmsgCI255`
  
  Changes information messages to blue

---

**-WmsgCU: RGB Color for User Messages**

**Group:**

MESSAGE

**Scope:**

Function

**Syntax:**

- `-WmsgCU <RGB>`

**Arguments:**

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

**Default:**

- `-WmsgCU0 (r00 g00 b00, black)`

**Description:**

With this option it is possible to change the user message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and specified in decimal. Hexadecimal needs a 0X prefix, (for gray errors: -WmsgCU0x808080).

**Example:**

- `-WmsgCU255`

  Changes user messages to blue
-WmsgCW: RGB Color for Warning Messages

Group:
MESSAGE

Scope:
Function

Syntax:
-WmsgCW <RGB>

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

Default:
-WmsgCW255 (r00 g00 bFF, blue)

Description:
With this option it is possible to change the warning message color. The value to be specified has to be a RGB (Red-Green-Blue) value, and specified in decimal. Hexadecimal needs a 0X prefix, (for gray errors: -WmsgCW0x808080).

Example:
-WmsgCW0
Changes warning messages to black

-WmsgFb: Set Message File Format for Batch Mode

Group:
MESSAGE

Syntax:
-WmsgFb [ v | m ]
Decoder Options
Using Decoder Options

Arguments:

\[ \text{v} : \text{Verbose format.} \]
\[ \text{m} : \text{Microsoft format.} \]

Default:

\[ -\text{WmsgFbm} \]

Description:

If the Decoder has been started with arguments, the Decoder operates in batch mode. There is no window visible and the Decoder terminates after job completion. In batch mode, messages are written to a file instead of the screen. This file only contains the messages.

By default, the Decoder uses a Microsoft message format to write messages (errors, warnings, information messages).

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 verbose message format may slow down decoding, because the decoder has to write more information into the message file.

Example:

None

-WmsgFi: Set Message Format for Interactive Mode

Group:

MESSAGE

Syntax:

\[ -\text{WmsgFi} [ \text{v} | \text{m} ]. \]

Arguments:

\[ \text{v} : \text{Verbose format} \]
\[ \text{m} : \text{Microsoft format} \]
### Decoder Options

#### Using Decoder Options

**Default:**

- `WmsgFiv`

**Description:**
If the Decoder is started without additional arguments (for example, files to be decoded), the Decoder is in interactive mode (that is, a window is visible). By default, the Decoder uses the verbose error file format to write the 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).

**NOTE**
Using the Microsoft format may speed up processing, because the compiler has to write less information to the screen.

**Example:**
None

---

**-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"`. 
Decoder Options
Using Decoder Options

Description:
With this option it is possible to modify the default message format in batch mode.
Following formats are supported (assume that the input file is x:\freescale\mysourcefile.object).

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%s</td>
<td>Source Extract</td>
<td>x:\freescale\sourcefile.object</td>
</tr>
<tr>
<td>%p</td>
<td>Path</td>
<td>x:\freescale\source</td>
</tr>
<tr>
<td>%f</td>
<td>Path and name</td>
<td>x:\freescale\source</td>
</tr>
<tr>
<td>%n</td>
<td>File name</td>
<td>mysourcefile</td>
</tr>
<tr>
<td>%e</td>
<td>Extension</td>
<td>.object</td>
</tr>
<tr>
<td>%N</td>
<td>File (8 chars)</td>
<td>mysource</td>
</tr>
<tr>
<td>%E</td>
<td>Extension (3 chars)</td>
<td>.obj</td>
</tr>
<tr>
<td>%l</td>
<td>Line</td>
<td>3</td>
</tr>
<tr>
<td>%c</td>
<td>Column</td>
<td>47</td>
</tr>
<tr>
<td>%o</td>
<td>Pos</td>
<td>1234</td>
</tr>
<tr>
<td>%K</td>
<td>Uppercase kind</td>
<td>ERROR</td>
</tr>
<tr>
<td>%k</td>
<td>Lowercase kind</td>
<td>error</td>
</tr>
<tr>
<td>%d</td>
<td>Number</td>
<td>M1815</td>
</tr>
<tr>
<td>%m</td>
<td>Message</td>
<td>text</td>
</tr>
<tr>
<td>%</td>
<td>Percent</td>
<td>%</td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
<td></td>
</tr>
</tbody>
</table>

Example:
None
-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%n\n%K %d: %m\n".

Description:
With this option, you can modify the default message format in interactive mode.
Following formats are supported (assume that the source file is
x:\hiware\mysourcefile.object):

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%s</td>
<td>Source Extract</td>
<td>x:\freescale\</td>
</tr>
<tr>
<td>%f</td>
<td>Path</td>
<td>x:\freescale\source</td>
</tr>
<tr>
<td>%n</td>
<td>File name</td>
<td>mysourcefile</td>
</tr>
<tr>
<td>%e</td>
<td>Extension</td>
<td>.object</td>
</tr>
<tr>
<td>%N</td>
<td>File (8 chars)</td>
<td>mysourcefile</td>
</tr>
<tr>
<td>%E</td>
<td>Extension (3 chars)</td>
<td>.obj</td>
</tr>
<tr>
<td>%l</td>
<td>Line</td>
<td>3</td>
</tr>
<tr>
<td>%c</td>
<td>Column</td>
<td>47</td>
</tr>
<tr>
<td>%o</td>
<td>Pos</td>
<td>1234</td>
</tr>
<tr>
<td>%K</td>
<td>Uppercase kind</td>
<td>ERROR</td>
</tr>
<tr>
<td>%k</td>
<td>Lowercase kind</td>
<td>error</td>
</tr>
<tr>
<td>%d</td>
<td>Number</td>
<td>M1815</td>
</tr>
<tr>
<td>%m</td>
<td>Message</td>
<td>text</td>
</tr>
<tr>
<td>%%</td>
<td>Percent</td>
<td>%</td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
<td></td>
</tr>
</tbody>
</table>
### -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 is not related to a specific file. Then this message format string is used. The following formats are supported:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%K</td>
<td>Uppercase kind</td>
<td>ERROR</td>
</tr>
<tr>
<td>%k</td>
<td>Lowercase kind</td>
<td>error</td>
</tr>
<tr>
<td>%d</td>
<td>Number</td>
<td>C1815</td>
</tr>
<tr>
<td>%m</td>
<td>Message</td>
<td>text</td>
</tr>
<tr>
<td>%</td>
<td>Percent</td>
<td>%</td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
<td></td>
</tr>
</tbody>
</table>

**Example:**

None
-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. For example, if a message is not related to a certain position then this message format string is used. Following formats are supported. Assume that the source file is:

x:\freescale\mysourcefile.object

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%p</td>
<td>Path</td>
<td>x:\freescale\</td>
</tr>
<tr>
<td>%f</td>
<td>Path and name</td>
<td>x:\freescale\source</td>
</tr>
<tr>
<td>%n</td>
<td>File name</td>
<td>mysourcefile</td>
</tr>
<tr>
<td>%e</td>
<td>Extension</td>
<td>.object</td>
</tr>
<tr>
<td>%N</td>
<td>File (8 chars)</td>
<td>mysource</td>
</tr>
<tr>
<td>%E</td>
<td>Extension (3 chars)</td>
<td>.obj</td>
</tr>
<tr>
<td>%K</td>
<td>Uppercase kind</td>
<td>ERROR</td>
</tr>
<tr>
<td>%k</td>
<td>Lowercase kind</td>
<td>error</td>
</tr>
<tr>
<td>%d</td>
<td>Number</td>
<td>D1815</td>
</tr>
<tr>
<td>%m</td>
<td>Message</td>
<td>text</td>
</tr>
<tr>
<td>%</td>
<td>Percent</td>
<td>%</td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
<td>\n</td>
</tr>
</tbody>
</table>

Example:

None
Decoder Options
Using Decoder Options

-WmsgNe: Number of Error Messages

Group:
MESSAGE

Syntax:
-WmsgNe <number>

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

Default:
50

Description:
With this option, you can set the number of error messages that occur before the Decoder stops.

Example:
-WmsgNe2
The Decoder stops processing after two error messages.

-WmsgNi: Number of Information Messages

Group:
MESSAGE

Syntax:
-WmsgNi <number>

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

Default:
50
Decoder Options
Using Decoder Options

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

Example:
-WmsgNi10
Only ten information messages are logged.

-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 that are not in the normal message categories
(WARNING, INFORMATION, ERROR, FATAL). With this option these messages can
be disabled. The idea of this option is to reduce the amount of messages and simplify error
 parsing of other tools.
a : Disables messages regarding included files.
b : Disables messages regarding reading files, for example, files used as input.
c : Disables messages regarding generated files.
d : Disables messages regarding statistics, for example, code size and RAM/ROM usage.
Decoder Options
Using Decoder Options

e : Disables informal messages, for example, memory model and floating point format messages.

**NOTE** Depending on the application, not all options may make sense. In this case they are ignored for compatibility.

Example:

```bash
-WmsgNu=c
```

**-WmsgNw: Number of Warning Messages**

**Group:** MESSAGE

**Syntax:**

```
-WmsgNw <number>
```

**Arguments:**

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

**Default:**

50

**Description:**

Use this option to set the number of warning messages.

**Example:**

```bash
-WmsgNw15
```

Only 15 warning messages are logged.

**-WmsgSd: Setting a Message to Disable**

**Group:** MESSAGE
Decoder Options
Using Decoder Options

Syntax:

-WmsgSd <number>

Arguments:

<number>: Message number to be disabled, for example 1801

Default:

None.

Description:

Use this option to disable a message, so it does not appear in error output.

Example:

None

-WmsgSe: Setting a Message to Error

Group:

MESSAGE

Syntax:

-WmsgSe <number>

Arguments:

<number>: Message number specified to be an error message, for example 1853

Default:

None.

Description:

Changes a message to an error message.

Example:

None
-WmsgSi: Setting a Message to Information

**Group:**
MESSAGE

**Syntax:**
-WmsgSi <number>

**Arguments:**
<number>: Message number specified to be an information message, for example 1853

**Default:**
None

**Description:**
Changes a message to an information message.

- WmsgSw: Setting a Message to Warning

**Group:**
MESSAGE

**Syntax:**
-WmsgSw <number>

**Arguments:**
<number>: Error number specified to be a warning message, for example 2901

**Default:**
None

**Description:**
Specifies a message to be a warning message.
Example:
None

-WOutFile: Create Error Listing File

Group:
MESSAGE

Syntax:
-WOutFile { On | Off }.

Arguments:
None

Default:
Error listing file is created

Description:
This option creates an error listing file. The error listing file contains a list of all messages and errors created during a compilation. Since the text error feedback can 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.
-WOutFileOff
No error file is created.

-WStdout: Write to Standard Output

Group:
MESSAGE
Decoder Options
Using Decoder Options

Syntax:
-WStdout ( On | Off )

Arguments:
None

Default:
Output written to stdout

Description:
With Windows applications, the standard streams are available. But text written into them
do not appear anywhere unless explicitly requested by the calling application. With this
option, text written to an error file is also written to stdout.

Example:
-WStdoutOn
All messages are written to stdout.
-WErrFileOff
Nothing is written to stdout.

-W1: No Information Messages

Group:
MESSAGE

Syntax:
-W1

Arguments:
None

Default:
None

Description:
Disables INFORMATION messages; only WARNING and ERROR messages are output.
-W2: No Information and Warning Messages

Group:
MESSAGES

Syntax:
-W2

Arguments:
None

Default:
None

Description:
Suppresses INFORMATION and WARNING messages, only ERRORs are printed.

-X: Write Disassembled Listing Only

Syntax:
-X

Arguments:
None

Default:
None

Description:
Writes the pure disassembly listing without any source or comments within the listing.
-Y: Write Disassembled Listing with Source And All Comments

Syntax:

\[-Y\]

Arguments:
None

Default:
None

File Format:
Only Freescale (former Hiware). ELF Object files are not affected by this option.

Description:

Writes the origin source and its comments within the disassembly listing.
Decoder Options

Using Decoder Options
Decoder Messages

This chapter describes messages produced by the application. Because of the huge amount of different messages produced, not all of them may be in this document.

Click any of the following links to jump to the corresponding section of this chapter:
- Types of Generated Messages
- Message Details
- List of Messages

Types of Generated Messages

Table 28.1 lists and describes the five types of generated messages.

<table>
<thead>
<tr>
<th>Message</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information</td>
<td>A message prints and compilation continues.</td>
</tr>
<tr>
<td>Warning</td>
<td>A message prints and processing continues. These messages indicate possible programming errors.</td>
</tr>
<tr>
<td>Error</td>
<td>A message prints and processing stops. These messages indicate illegal language usage.</td>
</tr>
<tr>
<td>Fatal</td>
<td>A message prints and processing aborts. These messages indicate a severe error, which causes processing to stop.</td>
</tr>
<tr>
<td>Disable</td>
<td>The message is disabled. No message is issued and processing continues. The application ignores the Disabled message.</td>
</tr>
</tbody>
</table>
Message Details

If an application generates a message, that message contains a message code as well as a four- to five-digit number. You can use this number to search for a message. The following message codes are supported:

- A = Assembler
- B = Burner
- C = Compiler
- D = Decoder
- L = Linker
- LM = Libmaker
- M = Maker

Messages that the application generates are sorted in ascending order for quick retrieval. Each message contains a description and usually a short example with a possible solution or tips to fix a problem.

For each message, the type of message is indicated. For example, [ERROR] indicates that the message is an error message.

[DISABLE, INFORMATION, WARNING, ERROR]

indicates that the message is a warning message by default, although you can change that message to DISABLE, INFORMATION or ERROR.

After the message type, there may be an additional entry indicating the language for which the message is generated:

- C++: Message is generated for C++
- M2: Message is generated for Modula-2
List of Messages

This section lists all messages. Message numbers less than 10000 are common to all tools. Not every compiler can issue all messages. For example, many compilers do not support any type of `struct` return. Those compilers will never issue the message C2500:

Expected: No support of class/struct return type.

D1:  Unknown Message Occurred

[FATAL]

Description:
The application tried to issue a message that was not defined. This is an internal error that should not occur. Please report this message to your distributor.

D2:  Message Overflow, Skipping <kind> Messages

(DISABLE, INFORMATION, WARNING, ERROR]

Description:
The application showed the number of message types specified with the options:

```
-WmsgNi: Number of Information Messages
-WmsgNw: Number of Warning Messages
-WmsgNe: Number of Error Messages
```

Additional messages of this type are not displayed.

**TIP** Use the options listed above to change the number of messages to display.

D50:  Input File ‘<file>’ Not Found

[FATAL]

Description:
The application was not able to find a file needed for processing.
TIP  Check if the file really exists. Check if you are using a file name containing spaces (in this case you have to quote it).

D51: 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 support statistic log files. Even if a tool does not support it, the message still exists, but is not issued in this case.

D52: Error in Command Line <cmd>

[FATAL]

Description
In case there is an error while processing the command line, this message is issued.

D64: Line Continuation Occurred in <FileName>

[DISABLE, INFORMATION, WARNING, ERROR]

Description:
In any environment file, the character '\', at the end of a line is interpreted as line continuation. This line and the next one are handled as one line. Because the path separation character of MS-DOS is also '\', paths that end with '\' are often incorrectly written. Instead, use a '.' after the last '\' unless you really want a line continuation.
Example:

Current Default.env:
...
LIBPATH=c:\freescale\lib
OBJPATH=c:\freescale\work
...
This is identical to:
...
LIBPATH=c:\freescale\libOBJPATH=c:\freescale\work
...
To fix it, append a '.' after the '\'
...
LIBPATH=c:\freescale\lib\.
OBJPATH=c:\freescale\work
...

D65: Environment Macro Expansion Message '<description>' for <variablename>

[DISABLE, INFORMATION, WARNING, ERROR]

Description:
During an environment variable macro substitution a problem occurred. Possible causes are that the named macro did not exist or a length limitation was reached. Also, recursive macros may cause this message.

Example:

Current variables:
...
LIBPATH=${LIBPATH}
...

TIP Check the definition of the environment variable.
Decoder Messages
List of Messages

D66: Search Path <Name> Does Not Exist

[DISABLE, INFORMATION, WARNING, ERROR]

Description:
The tool looked for a file that was not found. During the failed search for the file, a non existing path was encountered.

TIP  Check the spelling of your paths. Update the paths when moving a project. Use relative paths.

D1000: Bad Hex Input File <Description>

[DISABLE, INFORMATION, WARNING, ERROR]

Description:
While decoding a S-Record or an Intel Hex file, the decoder detected incorrect entries in the file. The <Description> gives more detail.

TIP  Check the descriptive text and if the file passed to the decoder is correct.

D1001: Because Current Processor is Unknown, No Disassembly is Generated. Use -proc.

[DISABLE, INFORMATION, WARNING, ERROR]

Description:
While decoding a S-Record or an Intel Hex file, the decoder needs to know about the processor used to decode the file with disassembly information. This is needed because these formats do not contain information about the processor.

TIP  Use the Option -Proc: Set Processor to specify the processor.
Decoder Controls

This chapter describes Decoder controls; pull-down menus and the Graphical User Interface (GUI).

Click any of the following links to jump to the corresponding section of this chapter:

- Pull-Down Menus
- Graphical User Interface
- Specifying the Input File
- Message and Error Feedback

Pull-Down Menus

The Decoder pull-down menus are on the menu bar of the Decoder main window (Figure 29.5). The following table lists and describes the main window’s top-level pull-down menus.

Table 29.1 Decoder Main Window Pull-down Menus

<table>
<thead>
<tr>
<th>Menu Name</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Options for managing configuration files</td>
</tr>
<tr>
<td>Decoder</td>
<td>Commands for setting options</td>
</tr>
<tr>
<td>View</td>
<td>Options for customizing window output</td>
</tr>
<tr>
<td>Help</td>
<td>Standard Windows Help menu</td>
</tr>
</tbody>
</table>
## Decoder Controls

### Pull-Down Menus

**File Menu**  
With the File pull-down menu (Figure 29.1), you can save or load configuration files. Configuration files contain:

- Configuration dialog option settings.
- Message settings that specify which messages to display and which to treat as errors.
- List of last commands executed and current command line
- Window position
- Tip of the Day settings, including if enabled at startup and current entry

**Figure 29.1 File Menu**

<table>
<thead>
<tr>
<th>File</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td></td>
</tr>
<tr>
<td>New / Default Configuration</td>
<td>Ctrl+N</td>
</tr>
<tr>
<td>Load Configuration</td>
<td>Ctrl+L</td>
</tr>
<tr>
<td>Save Configuration</td>
<td>Ctrl+S</td>
</tr>
<tr>
<td>Save Configuration As...</td>
<td>Ctrl+A</td>
</tr>
<tr>
<td>Configuration...</td>
<td></td>
</tr>
<tr>
<td>1 project.ini</td>
<td></td>
</tr>
<tr>
<td>Exit</td>
<td></td>
</tr>
</tbody>
</table>
The following table lists and describes the File menu selections:

Table 29.2 File Menu Selections

<table>
<thead>
<tr>
<th>Menu Selection</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td>Opens a standard <strong>Open File</strong> dialog. The selected file will be processed as soon as the open File box is closed using <strong>OK</strong>.</td>
</tr>
<tr>
<td>New/Default</td>
<td>Resets the option settings to the default value. The options which are activated per default are specified in section <strong>Command Line Options</strong>.</td>
</tr>
<tr>
<td>Configuration</td>
<td>Resets the option settings to the default value. The options which are activated per default are specified in section <strong>Command Line Options</strong>.</td>
</tr>
<tr>
<td>Load Configuration</td>
<td>Opens the standard open file dialog. Configuration data stored in the selected file is loaded and will be used by a further session.</td>
</tr>
<tr>
<td>Save Configuration</td>
<td>Saves the current settings.</td>
</tr>
<tr>
<td>Save Configuration as...</td>
<td>Opens a standard <strong>Save As</strong> dialog. The current settings are saved in a configuration file which has the specified name.</td>
</tr>
<tr>
<td>Configuration...</td>
<td>Opens the <strong>Configuration</strong> dialog to specify the editor used for error feedback and which parts to save with a configuration.</td>
</tr>
<tr>
<td>1..... project.ini</td>
<td>Recent project list. This list can be accessed to open a recently opened project again.</td>
</tr>
<tr>
<td>Exit</td>
<td>Closes the Decoder.</td>
</tr>
</tbody>
</table>
Decoder Controls
Pull-Down Menus

Decoder Menu

With the Decoder pull-down menu (Figure 29.2), you can customize the Decoder, graphically set or reset options, and access message settings.

Figure 29.2 Decoder Menu

The following table lists and describes the Decoder menu selections.

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Options</td>
<td>Displays the Option Settings dialog box, where you can define options for processing an input file.</td>
</tr>
<tr>
<td>Messages</td>
<td>Opens the Message Settings dialog box, where the different error, warning or information messages can be mapped to another message class.</td>
</tr>
</tbody>
</table>
View Menu

With the View Menu (Figure 29.3), you can customize the main window. You can specify whether the status bar and the tool bar are displayed or hidden, choose the font used in the window, and clear the window.

Figure 29.3 View Menu

The following table lists and describes the View menu selections.

Table 29.4 View Menu Selections

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool Bar</td>
<td>Displays tool bar in the main window.</td>
</tr>
<tr>
<td>Status Bar</td>
<td>Displays status bar in the main window.</td>
</tr>
<tr>
<td>Log...</td>
<td>Lets you customize the output in the main window content area.</td>
</tr>
<tr>
<td>Change Font</td>
<td>Opens a standard font-selection dialog. Your selections appear in the main window content area.</td>
</tr>
<tr>
<td>Clear Log</td>
<td>Lets you clear the main window content area.</td>
</tr>
</tbody>
</table>
Help Menu

From the Help pull-down menu (Figure 29.4), you can customize the Tip of the Day dialog and display help, as well as Decoder version and license information.

Figure 29.4 Help Menu

The following table lists and describes the Help menu selections.

Table 29.5 Help Menu Selections

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tip of the Day</td>
<td>Switches on or off the Tip of the Day display during startup.</td>
</tr>
<tr>
<td>Help Topics</td>
<td>Displays standard Help.</td>
</tr>
<tr>
<td>About...</td>
<td>Displays an About box with version and license information.</td>
</tr>
</tbody>
</table>

Graphical User Interface

This section describes important aspects of the Decoder graphical user interface (GUI). Windows and dialogs covered here are:

- Decoder Main Window
- Configuration Dialog
Decoder Controls
Graphical User Interface

Decoder Main Window
The Decoder main window displays if you do not specify a file name on the command line. If you start a tool using the Decoder, the Decoder main window does not appear.

Figure 29.5 Decoder Main Window

Main Window Components
The Decoder main window components are described in the sections that follow.

Window Title
The window title displays the tool name and project name. If no project is loaded, Default Configuration displays in the title area. An asterisk after the configuration name indicates you have made an unsaved change.

Tool Bar
Figure 29.6 indicates main window tool bar buttons.
Decoder Controls
Graphical User Interface

Figure 29.6 Decoder Main Window Tool Bar Buttons

The following table lists the Decoder main window tool bar buttons and describes their functions:

<table>
<thead>
<tr>
<th>Button Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Configuration</td>
<td>Same as the File &gt; New Configuration pull-down menu selection</td>
</tr>
<tr>
<td>Load a Configuration</td>
<td>Same as the File &gt; Load Configuration pull-down menu selection</td>
</tr>
<tr>
<td>Save the Current</td>
<td>Same as the File &gt; Save Configuration pull-down menu selection</td>
</tr>
<tr>
<td>Configuration</td>
<td></td>
</tr>
<tr>
<td>Online Help</td>
<td>Displays Decoder online help</td>
</tr>
<tr>
<td>Context Help</td>
<td>Changes cursor to question mark. When you hover your cursor over a</td>
</tr>
<tr>
<td></td>
<td>Decoder screen area and click the left mouse button, context-sensitive</td>
</tr>
<tr>
<td></td>
<td>help appears for the area you selected.</td>
</tr>
<tr>
<td>Command Line</td>
<td>Displays a context menu associated with the command line.</td>
</tr>
<tr>
<td>Decode</td>
<td>Starts execution of a desired command.</td>
</tr>
<tr>
<td>Stop Session</td>
<td>Stops the current session</td>
</tr>
<tr>
<td>Option settings</td>
<td>Displays the Option Settings dialog</td>
</tr>
<tr>
<td>Message settings</td>
<td>Displays the Message Settings dialog</td>
</tr>
<tr>
<td>Clear log</td>
<td>Clears main window content</td>
</tr>
</tbody>
</table>
Status Bar
The status bar (Figure 29.7) has two dynamic areas:

- Messages
- Time

When you point to a button in the tool bar or a menu entry, the message area displays the function of the button or menu entry.

The time field shows the start time of the current session (if one is active) or current system time.

Figure 29.7 Main Window Status Bar

Decoder Configuration Window
When you choose File > Configuration from the Decoder pull-down menus, the Configuration Window appears. The Configuration Window has three tabs:

- Editor Settings
- Save Configuration
- Environment
Editor Settings Tab

Figure 29.8 shows the Configuration Window with the Editor Settings tab selected.

Figure 29.8 Decoder Configuration Window - Editor Settings Tab
The following table lists and describes the Editor Settings tab controls.

### Table 29.7 Editor Settings Tab Controls

<table>
<thead>
<tr>
<th>Control</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Editor</td>
<td>Shared among all tools and projects on one computer. It is stored in the <code>MCUTOLS.INI</code> global initialization file.</td>
</tr>
<tr>
<td>Local Editor</td>
<td>Shared among all tools using the same project file</td>
</tr>
</tbody>
</table>
| Editor started with Command Line | Enable command-line editor. For the Winedit 32-bit version use the `winedit.exe` file  
                                         `C:\WinEdit32\WinEdit.exe%f /#:%l` |
| Editor started 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. |
| CodeWarrior (with COM)         | If selected, the CodeWarrior registered in the Windows Registry launches. |
| Editor Name                    | Type a name for the desired editor in the text-entry field               |
| Editor Executable              | Specify the editor’s path and executable name. Use the browse button (...) to locate the executable. |
| Editor Arguments               | Type in the command-line arguments for the editor in the text-entry field. Use %f for the filename, %l for the line number, and %c for the column number. |
Save Configuration Tab

Figure 29.9 shows the Configuration Window with the Save Configuration tab selected.

Figure 29.9 Decoder Configuration Window - Save Configuration Tab
Table 29.8 lists and describes the Save Configuration tab controls.

**Table 29.8  Save Configuration Tab Controls**

<table>
<thead>
<tr>
<th>Control</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Options</td>
<td>When checked, the current option and message settings are contained when a configuration is written. When unchecked, the last saved content remains valid.</td>
</tr>
<tr>
<td>Editor Configuration</td>
<td>When checked, the current editor setting is contained when a configuration is written. By disabling this option, the last saved content remains valid.</td>
</tr>
<tr>
<td>Appearance</td>
<td>When checked, window position, command line content, and history settings are contained when a configuration is written.</td>
</tr>
<tr>
<td>Environment Variables</td>
<td>When checked, the environment variable settings in the Environment Tab are written to the configuration.</td>
</tr>
<tr>
<td>Save on Exit</td>
<td>When checked, Decoder writes configuration on exit. No confirmation message appears. When unchecked, Decoder does not write configuration on exit, even if options or another part of the configuration has changed. No confirmation message appears when closing the Decoder.</td>
</tr>
</tbody>
</table>

**NOTE**  Settings are stored in the configuration file. Exceptions are recently used configuration list and settings in this dialog. Configurations can coexist in the same file as the shell project configuration. When the shell configures an editor, the Decoder can read the content from the project file. The shell project configuration filename is `project.ini`. 
Environment Tab

Figure 29.10 shows the Configuration Window with the Environment tab selected.

Figure 29.10 Decoder Configuration Window - Environment Tab

Use the Environment tab to configure the environment. The content of the tab is read from the project file in the [Environment Variables] section. You can choose from the following environment variables:

- General Path: GENPATH
- Object Path: OBJPATH
- Text Path: TEXTPATH
- Absolute Path: ABSPATH
- Header File Path: LIBPATH
- Various Environment Variables: other variables not covered in this list

Table 29.9 lists and describes the Environment tab controls.
### Table 29.9 Environment Tab Buttons

<table>
<thead>
<tr>
<th>Button</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Adds a new line/entry</td>
</tr>
<tr>
<td>Change</td>
<td>Changes a new line/entry</td>
</tr>
<tr>
<td>Delete</td>
<td>Deletes a new line/entry</td>
</tr>
<tr>
<td>Up</td>
<td>Moves up a line/entry</td>
</tr>
<tr>
<td>Down</td>
<td>Moves down a line/entry</td>
</tr>
</tbody>
</table>
Decoder Option Settings

The Options Settings Window displays when you select Decoder > Options from the pull-down menus. Click on the text in the list box to select an option. For help, select an option and press F1. The command-line option in the lower part of the dialog corresponds with your selection in the list box.

NOTE When options requiring additional parameters are selected, a dialog box or subwindow may appear.

Figure 29.11 Decoder Options Settings Window
Table 29.10 lists and describes the tabs in the Decoder Option Settings Window.

Table 29.10 Option Settings Window Tabs

<table>
<thead>
<tr>
<th>Tab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Command-line execution and print output settings</td>
</tr>
<tr>
<td>Input</td>
<td>Macro settings</td>
</tr>
<tr>
<td>Host</td>
<td>Lists options related to the host operating system</td>
</tr>
<tr>
<td>Messages</td>
<td>Message-handler settings - format, kind, and number of printed messages</td>
</tr>
</tbody>
</table>

**Messages Settings Window**

The Message Settings Window (Figure 29.12) displays when you select Decoder > Messages from the pull-down menus. This window lets you map messages to different message classes.

Each message has its own id (a character followed by a 4- or 5-digit number). This number allows you to search for the message in the manual and online help.

**Figure 29.12 Message Settings Window**
Table 29.11 lists and describes the tabs in the Message Settings window.

### Table 29.11  Message Settings Window Tabs

<table>
<thead>
<tr>
<th>Message Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disabled</td>
<td>Lists disabled messages. Messages displayed in the list box are not written to the output stream.</td>
</tr>
<tr>
<td>Information</td>
<td>Lists information messages. Information messages inform you of actions taken.</td>
</tr>
<tr>
<td>Warning</td>
<td>Lists warning messages. When a warning message is generated, processing of the input file continues.</td>
</tr>
<tr>
<td>Error</td>
<td>Lists error messages. When an error message is generated, processing of the input file stops.</td>
</tr>
<tr>
<td>Fatal</td>
<td>Lists fatal error messages. These messages report system consistency errors. Fatal error messages cannot be ignored or moved.</td>
</tr>
</tbody>
</table>

### Changing a Message Class

You can configure your own mapping of messages in different classes using one of the buttons at the right of the dialog. Each button refers to a message class. To change the class associated with a message, select the message in the list box and then click the button corresponding with the desired message class.

**Example:**

To define message D51 could not open statistic log file (warning message) as an error message:

1. Click the **Warning** tab
   
   A list of warning messages displays in the list box.

2. Click the string D51 could not open statistic log file in the list box.

3. Click the **Error** button to define the message as an error message.

**NOTE** You cannot move messages from or to the **fatal** error class.

**NOTE** The move to buttons are active only when you select messages that can be moved. When you try to move a message that can not be moved to a group, the corresponding move to button is greyed out.
If you want to validate the change you made in the error message mapping, click OK to close the Message Settings window. If you click the Cancel button, the previous message mapping remains valid.

Retrieving Information about an Error Message

You can access information about each message in the list box. Select the message in the list box, then click Help. An information box opens, which contains a more detailed description of the error message as well as a small example of code that could produce the error. If you select several messages, help for the first message displays. If you select no message, pressing F1 shows help for the dialog.

About Decoder Dialog Box

The About Decoder dialog box (Figure 29.13) displays when you select Help > About from the pull-down menus. This dialog box shows the current directory and the versions of Decoder components, with the version displayed at the top of the dialog box. Click OK to close the dialog box.

Figure 29.13 About Decoder Dialog Box
Specifying the Input File

There are different ways to specify the decode file to be processed. During processing, the software sets options according to configurations that you specified in Decoder windows. Before starting the decoding process of a file, use your editor to specify a working directory.

Use the Command Line in the Tool Bar to Decode

You can use the command line to process files. The command line lets you enter a new file name and additional Decoder options.

Processing a File Already Run

You can display the previously executed command using the arrow at the right of the command line. Select a command by clicking it, which puts it on the command line. The software processes the file you choose after you click the Decode button in the tool bar or press the Enter key.

File - Decode...

When you select File > Decode..., a standard open file dialog box displays. Browse to the file you want to process. The software processes the file you choose after you click the Decode button in the tool bar or press the Enter key.

Drag and Drop

You can drag a file from other programs (such as the File Manager or Explorer) and drop it into the Decoder main window. The software processes the dropped file after you release the mouse button.

If the dragged file has a .ini extension, it is loaded and treated as a configuration file, not as a file to be decoded.
Message and Error Feedback

After making, there are several ways to check for different errors or warnings. The format of an error message looks like this:

<msgType> <msgCode>: <Message>

Examples:

Could not open the file 'Fibo.abs'
FATAL D50: Input file 'Fibo.abs' not found
*** command line: 'Fibo.abs' ***
Decoder: *** Error occurred while processing! ***

The second example shows that messages from called applications are also displayed, but only if an error occurs. They are extracted from the error file if the application called has reported an error.

Using Information from the Main Window

Once a file has been processed, the Decoder window content area displays the list of detected errors or warnings. Use your editor of choice to open the source file and correct the errors.

Using a User-defined Editor

You must first configure the editor you want to use for message or error feedback in the Configuration dialog. Once a file has been processed, you can double-click on an error message. Your selected editor opens automatically and points to the line containing the error.
Decoder Controls
Message and Error Feedback
Maker: The Make Tool

This section describes the IDE Maker Utility. Maker implements the UNIX make command with a Graphical User Interface (GUI). In addition, you can use Maker to build Modula-2 applications as well as maintain C/C++ projects. Maker has:

- Online Help
- Flexible Message Management
- 32-bit functionality

This section consists of the following chapters:

- “Maker Controls”: Describes Maker controls, pull-down menus and the Graphical User Interface.
- “Using Maker”: Describes using Maker to build Modula-2 applications and to maintain C/C++ projects.
- “Maker Environment Variables”: Describes environment variables that Maker uses.
- “Building Libraries”: Describes how to use the Maker utility to adapt or build your own libraries.
- “Maker Options”: Describes options that control Maker process.
- “Maker Messages”: Describes Maker printed messages. Maker itself does not issue most of the messages generated in a make process, they usually result from other processes or programs started from Maker.

Starting the Maker Utility

All of the utilities described in this book may be started from executable files located in the Prog folder of your IDE installation. The executable files are:

- maker.exe Maker: The Make Tool
- burner.exe The Burner Utility
- decoder.exe The Decoder
- libmaker.exe Libmaker
- linker.exe The SmartLinker Utility
With a standard full installation of the HC(S)08/RS08 CodeWarrior IDE, the executable files are located here:

C:\Program Files\Freescale\CW08 v5.x\Prog

To start the Maker Utility, you can click on maker.exe.
Maker Controls

This chapter describes Maker controls, such as pull-down menus and the Graphical User Interface (GUI).

Click any of the following links to jump to the corresponding section of this chapter:

- Graphical User Interface
- Specifying the Input File
- Message and Error Feedback

Graphical User Interface

This section describes important aspects of Maker’s Graphical User Interface (GUI). This section covers these windows and dialogs:

- Maker Main Window
- Configuration Dialog

Maker Main Window

The Maker main window appears if you do not specify a file name on the command line. If you start a tool using Maker, the Maker main window does not appear.
Figure 30.1 Maker Main Window

Main Window Components
The Maker main window has these components:
- Window title
- Menu bar
- Tool bar
- Content area
- Status bar

Window Title
The window title displays the tool name and the project name. If Maker has no loaded project, Default Configuration appears in the title area. An asterisk after the configuration name indicates that you made an unsaved change.
Maker Main Window Menu Bar

Maker pull-down menus are on the menu bar of the main window. The following table lists and describes Maker’s top-level pull-down menus.

<table>
<thead>
<tr>
<th>Menu Name</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Selections for managing configuration files</td>
</tr>
<tr>
<td>Maker</td>
<td>Selections for setting options</td>
</tr>
<tr>
<td>View</td>
<td>Selections for customizing window output</td>
</tr>
<tr>
<td>Help</td>
<td>Standard Windows Help menu</td>
</tr>
</tbody>
</table>

**File Menu**

With the File Menu you can save or load configuration files. Configuration files contain:
- Configuration dialog option settings.
- Message settings that specify which messages to display and which to treat as errors.
- A list of the last command line executed and the current command line.
- The window position.
- Tips of the Day settings, including the startup settings and the current entry.

**Figure 30.2 File Menu**
The following table lists and describes **File** menu selections:

**Table 30.2 File Menu Selections**

<table>
<thead>
<tr>
<th>Menu Selection</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make</td>
<td>Opens a standard <strong>Open File</strong> dialog. Maker processes the selected file after you click <strong>OK</strong> to close the Open File dialog.</td>
</tr>
<tr>
<td>New/Default Configuration</td>
<td>Resets the option settings to default values. The <strong>Command Line Options</strong> section in this document specifies the default activated options.</td>
</tr>
<tr>
<td>Load Configuration</td>
<td>Opens the standard <strong>Open</strong> File dialog. Future sessions load and use the configuration data stored in the selected file.</td>
</tr>
<tr>
<td>Save Configuration</td>
<td>Saves the current settings.</td>
</tr>
<tr>
<td>Save Configuration as...</td>
<td>Opens a standard <strong>Save As</strong> dialog. Maker saves the current settings in a configuration file with the specified name.</td>
</tr>
<tr>
<td>Configuration...</td>
<td>Opens the <strong>Configuration</strong> dialog to specify the editor to use for error feedback and the parts to save with a configuration.</td>
</tr>
<tr>
<td>1.... project.ini</td>
<td>Recent project list. Access this list to open a recently used project again.</td>
</tr>
<tr>
<td>2....</td>
<td></td>
</tr>
<tr>
<td>Exit</td>
<td>Closes the Decoder.</td>
</tr>
</tbody>
</table>
**Maker Menu**

With the Maker Menu you can customize Maker, graphically set or reset options, and access message settings.

**Figure 30.3 Maker Menu**

![Maker Menu](image)

The following table lists and describes **Maker** menu selections.

**Table 30.3 Maker Menu Selections**

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Options</td>
<td>Displays the Options Settings dialog where you can define options for processing an input file.</td>
</tr>
<tr>
<td>Messages</td>
<td>Opens the Message Settings dialog where you can map error, warning, or information messages to different message classes.</td>
</tr>
<tr>
<td>Stop Making</td>
<td>Stops the current Make process. Maker grays out this selection when no active Make process exists.</td>
</tr>
</tbody>
</table>
**Maker Controls**  
*Graphical User Interface*

**View Menu**

With the View Menu you can customize the main window. You can choose the font used in the window, specify whether Maker displays or hides the status bar and the tool bar, and clear the window.

**Figure 30.4 View Menu**

![View Menu](image)

The following table lists and describes View menu selections.

**Table 30.4 View Menu Selections**

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool Bar</td>
<td>Toggles display of the tool bar in the main window.</td>
</tr>
<tr>
<td>Status Bar</td>
<td>Toggles display of the status bar in the main window.</td>
</tr>
<tr>
<td>Log...</td>
<td>Lets you customize the output in the main window content area.</td>
</tr>
<tr>
<td>Change Font</td>
<td>Opens a standard font-selection dialog. Your selections appear in the main window content area.</td>
</tr>
<tr>
<td>Clear Log</td>
<td>Lets you clear the main window content area.</td>
</tr>
</tbody>
</table>
Help Menu

From the Help Menu you can customize the Tip of the Day dialog. This menu can also display Windows help as well as Maker version and license information.

Figure 30.5 Help Menu

The following table lists and describes Help menu selections.

Table 30.5 Help Menu Selections

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tip of the Day</td>
<td>Toggles display of a Tip of the Day during startup.</td>
</tr>
<tr>
<td>Help Topics</td>
<td>Displays standard Help.</td>
</tr>
<tr>
<td>About...</td>
<td>Displays an About box with version and license information.</td>
</tr>
</tbody>
</table>
Maker Controls
Graphical User Interface

Maker Main Window Tool Bar

The Maker Main window tool bar icons are shown in Figure 30.6.

Figure 30.6 Maker Main Window Tool Bar Icons

The following table lists the Maker main window tool bar buttons and describes their functions.

Table 30.6 Main Window Tool Bar Icon

<table>
<thead>
<tr>
<th>Icon Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Configuration</td>
<td>Mimics the File &gt; New Configuration pull-down menu selection.</td>
</tr>
<tr>
<td>Load a Configuration</td>
<td>Mimics the File &gt; Load Configuration pull-down menu selection.</td>
</tr>
<tr>
<td>Save the Current</td>
<td>Mimics the File &gt; Save Configuration pull-down menu selection.</td>
</tr>
<tr>
<td>Configuration</td>
<td></td>
</tr>
<tr>
<td>Online Help</td>
<td>Displays Maker online help.</td>
</tr>
<tr>
<td>Context Help</td>
<td>Changes the cursor to a question mark. When you hover your cursor over a Maker screen area and click the left mouse button, context-sensitive help appears for the area you selected.</td>
</tr>
<tr>
<td>Command Line</td>
<td>Displays a context menu associated with the command line.</td>
</tr>
<tr>
<td>Make</td>
<td>Starts the execution of a desired command.</td>
</tr>
<tr>
<td>Stop Session</td>
<td>Stops the current session.</td>
</tr>
<tr>
<td>Option settings</td>
<td>Displays the Option Settings dialog.</td>
</tr>
<tr>
<td>Message settings</td>
<td>Displays the Message Settings dialog.</td>
</tr>
<tr>
<td>Clear log</td>
<td>Clears the main window content.</td>
</tr>
</tbody>
</table>
Maker Main Window Status Bar

The Maker Main window status bar has two dynamic areas:

- Messages
- Time

When you point to an icon on the tool bar or to a menu entry, the message area displays the function of the button or menu entry you point to.

The time field shows the start time of the current session (if an active session exists) or the current system time.

Figure 30.7 Main Window Status Bar

<table>
<thead>
<tr>
<th>Message Area</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ready</td>
<td>15:00:16</td>
</tr>
</tbody>
</table>

Maker Configuration Window

When you choose File > Configuration from the Maker Main window pull-down menus, the Configuration Window appears. The Configuration window has three tabs:

- Editor Settings
- Save Configuration
- Environment
Configuration Window Editor Settings Tab

Figure 30.8 shows the Configuration window with the Editor Settings tab selected.

Figure 30.8 Configuration Window - Editor Settings Tab
The following table lists and describes Editor Settings tab controls.

Table 30.7 Editor Settings Tab Controls

<table>
<thead>
<tr>
<th>Control</th>
<th>Function</th>
</tr>
</thead>
</table>
| Global Editor                     | Shared among all tools and projects on one computer. The
McuTools.INI global initialization file stores the global editor. |
| Local Editor                      | Shared among all tools using the same project file.                      |
| Editor started with Command Line  | Enable command-line editor starting. For the Winedit 32-bit version use the winedit.exe file
C:\WinEdit32\WinEdit.exe%f /#:%l |
| Editor started with DDE           | Enter the service, topic, and client name to use for a DDE connection to the editor. All entries can have modifiers for file name and line number. |
| CodeWarrior (with COM)            | If selected, the CodeWarrior version registered in the Windows Registry launches. |
| Editor Name                       | Enter in this text-entry field a name for the desired editor.             |
| Editor Executable                 | Specify the editor’s path and executable name. Use the browse button (...) to locate the executable file. |
| Editor Arguments                  | Enter in this text-entry field the command-line arguments for the editor. Use %f for the filename, %l for the line number, and %c for the column number. |

**NOTE:** Changing the Editor Selection option button settings in this window changes the entries in the text entry fields at the bottom of the window.
Configuration Window Save Configuration Tab

Figure 30.9 shows the Configuration dialog with the Save Configuration tab selected.

Figure 30.9  Configuration Dialog - Save Configuration Tab
The following table lists and describes the **Save Configuration** tab controls.

### Table 30.8  Save Configuration Tab Controls

<table>
<thead>
<tr>
<th>Control</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Options</td>
<td>When checked, Maker writes to the configuration file the current option and message settings. When unchecked, the last saved content remains valid.</td>
</tr>
<tr>
<td>Editor Configuration</td>
<td>When checked, Maker writes to the configuration file the current editor setting. When unchecked, the last saved content remains valid.</td>
</tr>
<tr>
<td>Appearance</td>
<td>When checked, Maker writes to the configuration file the window position, command-line content, and history settings. When unchecked, the last saved content remains valid.</td>
</tr>
<tr>
<td>Environment Variables</td>
<td>When checked, Maker writes to the configuration file the environment variable settings in the Environment Tab. When unchecked, the last saved content remains valid.</td>
</tr>
<tr>
<td>Save on Exit</td>
<td>When checked, Maker writes the configuration file on exit. No confirmation message appears. When unchecked, Maker does not write the configuration file on exit, even if you change options or another part of the configuration file. No confirmation message appears when closing Maker.</td>
</tr>
</tbody>
</table>

**NOTE**  
Maker stores settings in the configuration file, with exception of the recently used configuration list and the settings in this dialog. Configurations can coexist in the same file as the shell project configuration. When the shell configures an editor, Maker can read the content from the project file. The shell project configuration filename is `project.ini`. 

---

*HC(S)08/RS08 and HC(S)12 Build Tools Utilities Manual*  
603
Configuration Window Environment Tab

Use the Configuration window with the Environment tab selected to configure the environment.

Figure 30.10 Configuration Dialog - Environment Tab
Maker reads the content of the dialog from the [Environment Variables] section of the actual project file. You can choose from these environment variables:

- General Path: GENPATH
- Object Path: OBJPATH
- Text Path: TEXTPATH
- Absolute Path: ABSPATH
- Header File Path: LIBPATH
- Various Environment Variables: other variables not covered in this list

The following table lists and describes the Configuration window Environment tab controls.

**Table 30.9 Environment Tab Buttons**

<table>
<thead>
<tr>
<th>Button</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Adds a new line/entry.</td>
</tr>
<tr>
<td>Change</td>
<td>Changes a new line/entry.</td>
</tr>
<tr>
<td>Delete</td>
<td>Deletes a new line/entry.</td>
</tr>
<tr>
<td>Up</td>
<td>Moves up a line/entry.</td>
</tr>
<tr>
<td>Down</td>
<td>Moves down a line/entry.</td>
</tr>
</tbody>
</table>
Tip of the Day Window

When you start the tool, a Tip of the Day window displays a randomly chosen user tip.

Figure 30.11  Tip of the Day Window

The Next Tip button lets you read the next hint. If you don’t want the Tip of the Day window to open after the program starts, uncheck the Show Tips on StartUp box. Click Close to close the Tip of the Day window.

NOTE  The local project file stores user configurations.
Maker Options Settings Window

The Options Settings window appears when you select Maker > Options from the pull-down menus. Click once on the text in the list box to select an option. For help, select an option and press F1. The command-line option in the lower part of the dialog corresponds to your selection in the list box.

NOTE When you select options requiring additional parameters, a dialog box or subwindow may appear.

Figure 30.12 Options Settings Window
The following table lists and describes the tabs in the **Option Settings** dialog.

### Table 30.10  Option Settings Dialog Tabs

<table>
<thead>
<tr>
<th>Tab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Command-line execution and print-output settings.</td>
</tr>
<tr>
<td>Input</td>
<td>Macro settings.</td>
</tr>
<tr>
<td>Host</td>
<td>Lists options related to the host operating system.</td>
</tr>
<tr>
<td>Messages</td>
<td>Message-handler settings, such as format, kind, and number of printed</td>
</tr>
<tr>
<td></td>
<td>messages.</td>
</tr>
<tr>
<td>Modula-2</td>
<td>Modula-2 make-specific options (not relevant for C users).</td>
</tr>
</tbody>
</table>
Maker Message Settings Window

The **Message Settings** window appears when you select **Maker > Messages** from the pull-down menus. This window lets you map messages to different message classes.

Each message has its own ID (a character followed by a 4- or 5-digit number). This number allows for message look-up both in the manual and in the online help.

Figure 30.13  Message Settings Window
The following table lists and describes the tabs in the Message Settings window.

### Table 30.11 Message Settings Window Tabs

<table>
<thead>
<tr>
<th>Message Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disabled</td>
<td>Lists disabled messages. Maker does not write to the output stream the messages displayed in the list box.</td>
</tr>
<tr>
<td>Information</td>
<td>Lists information messages. Information messages inform you of actions taken.</td>
</tr>
<tr>
<td>Warning</td>
<td>Lists warning messages. When Maker generates a warning message, it continues processing the input file.</td>
</tr>
<tr>
<td>Error</td>
<td>Lists error messages. When Maker generates an error message, it stops processing the input file.</td>
</tr>
<tr>
<td>Fatal</td>
<td>Lists fatal error messages. These messages report system consistency errors. You cannot ignore or move fatal error messages.</td>
</tr>
</tbody>
</table>

### Changing a Message’s Class

You can configure your own mapping of messages in different classes using one of the buttons at the right of the dialog. Each button refers to a message class. To change the class associated with a message, select the message in the list box and then click the button corresponding with the desired message class.

**Example:**

To define message M5116 could not delete file (a warning message) as an error message, follow these steps:

1. Click the Warning tab
   
   A list of warning messages appears in the list box.
2. Click the string M5116 could not delete file in the list box.
3. Click the Error button to define the message as an error message.

**NOTE** You cannot move messages from or to the fatal error class. Maker only enables the move to buttons when you select movable messages. If you try to move a message to an impermissible group, Maker grays out the impermissible move to button.
If you want to validate the modification you performed in the error message mapping, click **OK** to close the **Message Settings** dialog. If you click **Cancel** to close the dialog, the previous message mapping remains valid.

**Retrieving Information about an Error Message**

You can access information about each message in the list box. Select the message in the list box, then click **Help**. An information box opens, which contains a more detailed description of the error message as well as a small example of code producing it. If you select several messages, help for the first message appears. If you select no message, pressing **F1** shows the help for the dialog.
About Dialog Box

The About dialog box appears when you select Help > About from the pull-down menus. This dialog shows the current directory and the Maker component versions. The Maker version appears separately at the top of the dialog. Click OK to close the dialog.

**NOTE** During a Maker process, Maker component subversions do not appear. Maker must be idle in order for subversions to appear.

![Figure 30.14 About Dialog Box](image-url)
Specifying the Input File

There are different ways to specify the make file to process. During processing, the software sets options according to the configurations that you specified in Maker dialogs. Before starting to process a make file, you specify a working directory using your editor.

Use the Command Line in the Tool Bar to Make

This section shows how to use the command line to process files. The command line lets you enter a new file name and additional Maker options.

Processing a File Already Run

You can display the previously executed command using the arrow at the right of the command line. Select a command by clicking it, which puts it on the command line. The software processes the file you choose after you click the Make button in the tool bar or after you press the Enter key.

File - Make...

When you select File > Make... from the pull-down menus, a standard Open File dialog appears. Navigate and select the file you want to process. The software processes the file you choose after you click the Make button in the tool bar or press the Enter key.

Drag and Drop

You can drag a file from other software (such as the File Manager or Explorer) and drop it into the Maker main window. The software processes the dropped file just after you release the mouse button.

If the dragged file has the ini extension, Maker loads and treats it as a configuration file, not as a makefile. To process a makefile with an ini extension, use another method to run it.
Message and Error Feedback

After making, there are several ways to check where Maker detected different errors or warnings. The format of an error message looks like this:

<msgType> <msgCode>: <Message>

Examples:
ERROR M5102: input file not found
ERROR M5112: called application: "ERROR C1011: Undeclared enumeration tag"

The second example shows that Maker also displays messages from called applications, but only if an error occurs. Maker extracts the messages from the error file if the called application reports an error.

Using Information from the Main Window

After Maker processes a file, the Maker window content area displays a list of detected errors or warnings. Use your editor of choice to open the source file and correct the errors.

Using a User-defined Editor

You must first configure the editor you want to use for message or error feedback in the Configuration dialog. After Maker processes a file, you need only double-click an error message to have your selected editor open automatically and point to the line containing the error.
Using Maker

With Maker you can build Modula-2 applications as well as maintain C/C++ projects. Maker syntax is a subset of the UNIX Make command.

Click any of the following links to jump to the corresponding section of this chapter:
- Making Modula–2 Applications
- Making C Applications
- User-defined Macros (Static Macros)
- Directives and Special Targets

Making Modula–2 Applications

To make a Modula-2 application, enter the name of the main module at the input prompt (or the command line). First, Maker collects dependencies given by the IMPORT clauses in the source files of both implementation and definition modules. Second, Maker recompiles files modified since the last compilation. Third, Maker tries to link the application.

The Make utility needs three environment variables:
- LINK: Linker for Modula–2 — Defines the linker program
- COMP: Modula–2 Compiler — Defines the compiler
- FLAGS: Options for Modula–2 Compiler — Defines the compiler options for the compiler given in COMP

These variables are necessary only when you use the Maker to build a Modula–2 application. For makefile processing, they are not needed (although you can use them as macros, as described later in this chapter).

Making C Applications

Since in C you cannot necessarily deduce dependencies between files by looking at the source files, automatic make (as with Modula-2 applications) is not possible. However, if you describe these dependencies in a file, Make can process this so-called makefile and (re-)build a C application.
Using Makefiles

This section gives a short introduction to writing and using makefiles. If you already know UNIX–style make utilities, you probably already know most of what follows. If you have been working until now with Microsoft Make, we strongly recommend that you read this section.

Syntax of Makefiles

Makefile syntax is as follows:

MakeFile  =  {Entry | Directive}.
Entry     =  {Macro | Update | Rule}.
Macro     =  Name (*=""|"+="|"=+" ) Line NL.
Update    =  Name ":=" [Name {[","] Name}] NL (Command).
Command   =  WhiteSpace (WhiteSpace) Line NL.
Rule      =  "." Suffix ["." Suffix] ":=" NL (Command).
Directive =  INCLUDE Name NL.
WhiteSpace = " "|"	".
NL        = 
Line      =  {<any char except un-escaped linebreaks> }.
Name      =  <any valid file name>.
Letter    =  any letter from "A" to "Z" or from "a" to "z" .

Case Sensitivity

By default, Maker is case-sensitive. However, if you set the –C option, Maker treats uppercase and lowercase letters the same.

Line Breaks

Processing a makefile is a line–oriented job because you use a linebreak to terminate most constructs such as macro definitions or dependency lists. If you want Make to ignore a linebreak, escape it by placing a backslash (“\”) immediately before the linebreak. Make then reads the combination of backslash and linebreak as one single blank. You cannot use a line continuation to enlarge comment lines.
Comments

Comments in a makefile start with the number sign (#) and end with the next linebreak.

Dependencies

Makefile update entries determine dependencies between files. Such an update entry has the form:

target file: {dependency file} {command line}

This entry tells HI-CROSS Make that the target file depends on all the dependency files. If any of the dependency files changed since the last target-file make, or if the target file does not exist, Make executes the command lines in order of appearance. If dependencies do not exist, Make always executes the command lines. If command lines do not exist, the target needs re-making, and rules are inapplicable, Make issues an error message. See the following sections for more information on rules.

Commands

You must begin each command on a new line and prefix that command by at least one blank or tab. Maker does not claim the tab as in UNIX make. The following list describes additional characteristics:

- Maker strips leading and trailing blanks and tabs from the command line.
- If the command line terminates with an exit code not equal to zero, Maker displays an error message and stops makefile processing, unless the line starts with a dash (–). Maker removes the dash before executing the command.
- A star (*) at the start of the command line prevents Maker from capturing the output of the called tool. Sixteen-bit applications such as command.com especially need this star.

Processing

Make processes updated entries recursively, which means that if a dependency file appears as a target in some other update entry, Make processes that other update entry first. If a dependency file does not exist and rules are inapplicable, Make issues an error message. See the following sections for more information on rules.

Normally, makefile processing starts with the update entry for the target given on the command line or at the input prompt. If you do not specify a target, processing starts at the first update entry in the makefile.

If there are two update entries for the same target file, Make appends the dependencies and commands of the second update entry to those of the first update entry. Make issues an error message if it finds circular dependencies.
Using Maker
User-defined Macros (Static Macros)

Macros

Macros associate a name with some arbitrary text. You can substitute this name for each occurrence of the arbitrary text in the makefile. There are two different forms of macros: user defined static ones and predefined dynamic ones.

User-defined Macros (Static Macros)

This section describes the form of a macro definition.

Definition

A macro definition has the form:

\[ \text{macro\_name} = \text{text up to the next un-escaped linebreak} \]

After you define a macro, you can use a macro reference to include the text at any place in a makefile.

Reference

A macro reference has the form:

\[ \$(\text{macro\_name}) \]

Make will wholly replace the reference with the text, including the “\$ (“ and the “) ”. If the text itself contains more macro references, Make expands those, too.

Redefinition

You can redefine macros, in which case the text in the new definition overwrites the text in the old definition. Maker issues an error message if it detects a circular macro definition like this:

\[ \text{ThisMacro} = \$(\text{ThatMacro}) \]
\[ \text{ThatMacro} = \text{Not } \$(\text{ThisMacro}) \]

Macro Substitution

During macro expansion, use the following syntax to have Maker replace strings:

\[ \$(\text{macro\_name}:\text{find}=\text{replace}) \]

In this example, Maker replaces every occurrence of “find” with “replace”. 

Use this kind of macro expansion to derive filenames, as in the following example:

```
SRCNAMES = a.c b.c
OBJNAMES = $(SRCNAMES:.c=.o)
```

The result of this example is that OBJNAMES contains “a.o b.o”.

**NOTE**  Maker does not allow spaces in the search string, the replace string, the whole macro definition, or before or after the “:” or the “=”

### Macros & Comments

If a comment follows a macro on the same line, as in the following example, the text that replaces any reference of these macros ends just before the # character:

```
MyMacro   = another #And that’s a comment
OurMacro  = This is \
$(MyMacro) example #That’s a comment, too!
MyMacro   = a third #Redefinition of a macro
HisMacro  = This is \
  $(MyMacro) example
```

Maker replaces the macro references as follows (without double quotes):

```
$(MyMacro) = "a third"
$(OurMacro) = "This is a third example"
$(HisMacro) = "This is a third example"
```

You can use macro references in update entries, inference rules, macro definitions, and macro references. See the following sections for more information on rules. The macro-reference possibility allows constructs such as:

```
This = Macro
MyMacro = This is a circular macro reference!
$(My$(This))
```

This example first evaluates to “$(MyMacro)" and then to “This is a circular macro reference!”.
Using Maker
User-defined Macros (Static Macros)

Concatenation

Besides the macro definition operator "=", Make knows two additional operators: "+=" and "=+. The first operator appends the text on the right to the macro on the left. The second operator assigns to the macro the value given by appending the macro's previous value to the text given on the right:

```
MyMacro = File
MyMacro += .TXT
#Now the macro has the value "File.TXT"
MyMacro =+ D:\
#Now it has the value "D:\File.TXT"
```

The following macro is a case handled differently by different make utilities. For example, in HI-CROSS Make it has the value "D:\SomeDir\". Some other make implementations would expand it as "D:\SomeDir" and take the last backslash as part of an escaped linebreak:

```
MyMacro = D:\SomeDir\n```

Command-Line Macros

There are two kinds of User Defined Macros: Command-line macros and makefile macros. Makefile macros are the macro definitions that appear in the make file. Command-line macros are macros on the command line with option -d. Command-line macros have a higher priority than macros defined in the makefile or in a file included. Therefore, if you define a macro on command line, Maker ignores further definition of a macro with the same name in the makefile.

A special command-line macro is TARGET, which defines the name of the top target to make. The TARGET macro provides compatibility with previous Maker versions. Specify a top target by adding its name after the makefile name. Defining an explicit top target with the TARGET macro works only on the command line. The TARGET macro in the makefile does not define a new top target. Do this explicitly by specifying a new target at the top, which has the top target to make as dependency.
**Dynamic Macros**

In addition to user–defined macros, which are always static, **HI-CROSS Make** recognizes the following dynamic macros, which differently evaluate in different contexts:

- $*$ base name (without suffix and period) of the target file.
- $@$ complete target file name.
- $<$ complete list of dependency files.
- $? list of dependency files that are younger than the target.
- $$ evaluates to a single dollar sign.

Except for the first one and the last one, these dynamic macros may only appear within command lines. Maker replaces them at the very end of macro substitution, just as it executes the command:

```
MyMacro = $<
OurMacro = file.c $(MyMacro)
THAT.EXE : $*.C $(OurMacro)
 $(COMP) $(MyMacro)
 $(LINK) $*.PRM
```

In a first step, the first line evaluates to:

```
THAT.EXE : THAT.C file.c $<
```

This line is circular, since Maker will now replace $< with "$THAT.C file.c $<" and so on. For this reason, the dynamic macros $< and $? may only appear on a command line (after Maker completes all macro substitution). If we now assume that `OurMacro` was defined as:

```
OurMacro = file.c io.c
```

after Maker completes all macro substitution, we get:

```
THAT.EXE : THAT.C file.c io.c
```

Example of $<:

```
target.o: target.c a.c b.c
 $(COMP) $<
```

replaced with:

```
target.o: target.c a.c b.c
 $(COMP) target.c a.c b.c
```

Example of $?:

```
target.o: target.c a.c b.c
 $(COMP) $?
```
If a.c and b.c are newer than target.o, then the result will be:

target.o: target.c a.c b.c

$(COMP) a.c b.c

NOTE HI-CROSS Make also defines macros for all environment variables currently set, which you can redefine like any other macro.

Inference Rules

Inference rules specify default rules for certain common cases. Inference rules have the form:

.depSuffix.targetSuffix:
    {Commands}

or:

.depSuffix:
    {Commands}

These rules tell HI-CROSS Make how to make a file with suffix targetSuffix if it cannot find an update entry for the file: look for a file with the same name as the target but with suffix depSuffix. Assume the target depends on that file, make the usual checks, and if Maker has to remake the target, execute the commands. If commands do not exist and the target needs remaking, Maker issues an error.

The second form of an inference rule with only one suffix works exactly as the first one. Maker assumes an empty target suffix.

For example, object files usually depend on a source file of the same name, but with a different suffix, and Make calls a compiler to create those object files. If we assume that object files have the extension .o and source files have the extension .c, we could express this example as:

.c.o:
    $(COMP) $*.c

If Make now finds a dependency file with extension .o (for example, THIS.o) but no update entry having this file as target, it applies the above rule. The result is exactly the same as if your makefile contained the dependency:

THIS.o: THIS.c
    $(COMP) $*.c

Rules also play a different role: if there is an update entry without command lines, HI-CROSS Make searches for a rule that might apply and executes the commands
specified in that rule. For example, with your makefile containing above rule, the update entry:

```
THAT.o: FILE.h DATA.h
```

This is equivalent to:

```
THAT.o: FILE.h DATA.h THAT.c

$(COMP) $*.c
```

If you define two different inference rules for the same target suffix, only the last one is active.

If HI-CROSS Make finds a dependency file that does not appear as a target in some other update entry, it tries to find an inference rule to apply. If that try fails too, and the file exists, Make assumes that the file is up to date. If the file does not exist, Maker needs to remake it. Since Maker lacks a rule or an update entry for the file, it issues an error message.

Here is a more complex example:

```make
# demo make file for assembly project

OBJECTS = a_1.o a_2.o a_3.o
ASM = c:\freescale\prog\assembler.exe
LINK = c:\freescale\prog\linker.exe
all: myasm.abs
   echo "all done"
myasm.abs: $(OBJECTS) myasm.prm
a_1.o: a_1.inc
a_2.o: a_1.inc a_2.inc
   .prm.abs:
   $(LINK) $*.prm
   .asm.o :
   $(ASM) $*.asm
```

### Multiple Inference Rules

You can specify more than one inference rule for each dependency suffix. Use this technique when you have source files written in different programming languages with different file suffixes. For example, assume you have sources written in assembly language, in Anise-C and C++. The object files produced by the assembler and compiler have all the same suffixes. They are linked together to one program or library. You can represent this relationship by one target having all the object files as a dependency list:

```
makeAll: asm_obj1.o asm_obj2.o asm_obj3.o c_obj1.o cobj2.o cpp_obj1.o
```
Using Maker

Directives and Special Targets

These rules build the object files:

```
.asm.o:
  $(ASSEMBLE) $*.asm $(ASMOPTIONS)
.c.o:
  $(COMPILE) $*.c $(COPIONS)
.cpp.o:
  $(COMPILE) $*.cpp $(CPPOPTIONS)
```

Maker selects the first applicable rule.

**NOTE**

The Maker resolution algorithm is logically incomplete. You can chain rules together in some cases, but doing so may lead to conflicts with the handling of multiple inference rules. For example, if you use template frames with the suffix .tpl compiled by a program that produces C files from TPL files, Maker may have problems resolving multiple rules in the further compilation steps. Tip: Construct and use a test makefile that contains the main resolution features in order to investigate Maker’s build behavior. If the test makefile works, the full makefile also works.

Directives and Special Targets

**HI-CROSS Make** lets you include one makefile into another by using an include directive of the form:

```
INCLUDE filename
```

This directive textually replaces the include directive with the given file’s contents (from another makefile). If Make cannot locate, open, or read the file, it issues an error message.

Make always includes the default makefile DEFAULT.MAK at the very beginning. The environment variable GENPATH specifies the directory that contains the makefile.

**NOTE**

Because the DEFAULT.MAK is included automatically, you have to be careful when using this name. An illegal DEFAULT.MAK does cause failures in all other makefiles for which it is in the search path. We do recommend to share common definitions by explicit makefile includes instead of using the implicitly included DEFAULT.MAK.

Make issues an error message for circular includes.

**HI-CROSS Make** also allows definition of two special targets without dependencies:

```
BEFORE:
  {Commands}
```
and
AFTER:

({Commands})

Make executes these commands just before and just after processing the top target given on the command line.

**Built-In Commands**

You can start DOS programs from the HI-CROSS Make Utility on the command line. You can directly execute external DOS commands; to execute built-in commands call COMMAND.COM with option /c, like this:

*COMMAND.COM /c dir C:\freescale > C:\DIR.TXT

**NOTE**  The star (“*”) prevents Maker from capturing the output of command.com. The output capture facility sometimes has problems with 16-bit executables like command.com. In WinNT environments, use the native 32 bit shell “cmd.exe” instead of command.com.

The HI-CROSS Make Utility also has a few simple built-in commands. Subsequently, we’ll discuss each of these commands.

**copy file1 file2**

This command creates a copy of file1 with the name file2. No wildcards are allowed. If you need wildcards, use the DOS built-in copy command.

**del file1 file2... fileN**

This command deletes the files passed as arguments. Again, no wildcards are allowed. Maker follows the file path from the current directory, if you do not specify an absolute path. Maker does not consult the environment settings to find the files to delete.

**cd directory**

This command changes temporarily to the current directory. The scope of the ‘cd’ command is the command list of a target where Maker called it.

**NOTE**  Avoid the use of this command whenever possible. The command may lead to inconsistency with relative-path definitions in the environment.

**echo text**

This command is actually a no-op. If Maker displays the commands, it displays the text, too. You can view the echo command as a way of defining a comment that Maker shows, while hiding normal comments starting with “#”.

**puts outputfile text**
Using Maker
Directives and Special Targets

This command writes text, the rest of the command line, to the file specified with outputfile (the first identifier of the command line). The write mode is appending. If the file does not exist, Maker creates it (mode “a+”).

Example:
puts myOutput.txt This is a text
This example writes the text “This is a text” with a line break at the end to the file myOutput.txt.

Example:

GENMAKE= bb.mak
TARGET = b
MAKE= c:\freescale\prog\maker.exe
COMP= c:\freescale\prog\compiler.exe
STAR=* 
DEPENDENDS = $(TARGET).c $(TARGET).h
create$(GENMAKE):
  -del $(GENMAKE)
  puts $(GENMAKE) \nCOMP=$(COMP)
  puts $(GENMAKE) \nMAKE=$(MAKE)
  puts $(GENMAKE) \n $(TARGET).o : $(DEPENDENDS)
  puts $(GENMAKE) \n $$($(COMP) $(TARGET).c
  $(MAKE) $(GENMAKE) $(TARGET).o

This example generates and runs bb.mak:

COMP=c:\freescale\prog\compiler.exe
MAKE=c:\freescale\prog\maker.exe
b.o : b.c b.h 
  c:\freescale\prog\compiler.exe b.c
fc file1 file2

This example compares two files, specified by name as file1 and file2, byte by byte and remembers the result for the next '?' command. The result is TRUE if the files are identical and FALSE if they are not identical.

text file1 file2

This example compares two text files byte by byte, ignoring blanks for compare, and remembers the result for the next '?' command. The result is TRUE if the files are identical and FALSE if they are not identical.

Syntax:   ? <commandIfYes> ‘:’ <commandIfNo>
The result of the last compare operation does cause either `<commandIfYes>` to be executed if the compared files were identical or `<commandIfNo>` if the compared files were not identical.

**Example:**

```
fctext  upxcall.c upxcall.old
? puts log.txt files are equal : puts log.txt files are not equal
```

or written in two lines

```
fctext  upxcall.c upxcall.old
? puts log.txt files are equal \  
: puts log.txt files are not equal

rehash
```

This example re–loads the HI-CROSS environment from the default.env file. Thereafter all commands, all macro expansions, and all file searches execute in the new environment.

```
ren file1 file2
```

This example renames file1 to file2. No wildcards are allowed.

**Command Line**

The Maker command line consists of three parts:

- **Maker Options**
  Maker treats all entries starting with a dash (‘-’) as options. To specify the top target, use the target name on the command line after the makefile name.

- **Makefile name**
  Maker treats the first command line argument, which does not start with a dash, as a makefile name.

- **Targets**
  Maker treats all remaining arguments without a leading dash as targets to build. If you do not specify targets, the first rule is build.

When you start Maker without command-line arguments, a window opens in which you can manually enter commands.

**Implementation Restrictions**

**Make** has only one implementation restriction: the string resulting from a macro substitution cannot contain more than 4095 characters.
Maker Environment Variables

This section describes environment variables that Maker uses. Other tools also use some of these environment variables.

Click any of the following links to jump to the corresponding section of this chapter:

- Setting Parameters
- Input and Output Files
- List of Environment Variables

Setting Parameters

Set Maker parameters in an environment using environment variables. The syntax is always the same:

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

**NOTE**

No blanks are allowed in an environment-variable definition.

**Example:**

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

Maker parameters may be defined:

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

**NOTE**

The maximum length of environment-variable entries in the `DEFAULT.ENV/ .hidefaults` is 1024 characters
NOTE You can set the default directory mentioned above using 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 the programs cannot find a definition, they use a default value.

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 (such as for the DEFAULT.ENV/.hidefaults).

Normally, the operating system or the program that launches another program (for example, WinEdit) determines the current directory of a started tool.

For the UNIX operating system, the current directory of an executable started is also the current directory from where the binary file started.

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

- If the File Manager or Explorer launches the tool, the current directory is the location of the launched executable.
- If clicking icon on the desktop launches the tool, the current directory is the one specified and associated with the icon.
- If dragging a file on the icon of the executable file under Windows 95 or Windows NT 4.0 launches the tool, the current directory is the desktop.
- If another launching tool with its own current directory specification (such as the WinEdit editor) launches the tool, the current directory is the one specified by the launching tool (such as the current directory definition in WinEdit).
- If the tool supports local project files, the current directory is where the local project file resides. Changing the current project file also changes the current directory, if the other project file is in a different directory. Note that browsing for a C file does not change the current directory.

To override this behavior, you can use the environment variable DEFAULTDIR. The current directory appears among other information with the maker option “-v” and in the about box.
Global Initialization File (MCUTOOLS.INI)

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

Example

C: \ WINDOWS \ MCUTOOLS.INI
D: \ INSTALL \ PROG \ MCUTOOLS.INI

If a tool starts in the D: \ INSTALL \ PROG \ DIRECTOY, it uses the project file in the same directory as the tool:
(D: \ INSTALL \ PROG \ MCUTOOLS.INI).

If the tool starts outside the D: \ INSTALL \ PROG directory, it uses the project file in the Windows directory:
(C: \ WINDOWS \ MCUTOOLS.INI).

The following section gives a short description of the entries in the MCUTOOLS.INI file.

[Installation] Section

Entry:
Path

Arguments:
Last installation path

Description:
Whenever you install a tool, the installation script stores the installation destination directory in this variable.

Example:
Path=c: \ install

Entry:
Group
**Maker Environment Variables**

*Setting Parameters*

---

**Arguments:**

Last installation program group.

**Description:**

Whenever you install a tool, the installation script stores the created program group in this variable.

**Example:**

`Group=Make`

---

**[Options] Section**

**Entry:**

DefaultDir

**Arguments:**

Default Directory to use.

**Description:**

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

**Example:**

`DefaultDir=c:\install\project`

---

**[Editor] Section**

**Entry:**

Editor_Name

**Arguments:**

The name of the global editor
Maker Environment Variables
Setting Parameters

**Description:**
Specifies the name displayed in the global editor. This entry has only a descriptive effect. Its content does not apply to starting the editor. Maker cannot modify this entry.

**Entry:**
Editor_Exe

**Arguments:**
The name of the executable file of the global editor

**Description:**
Specifies file name 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 exists and is not empty. Maker cannot modify this entry.

**Entry:**
Editor_Opts

**Arguments:**
The options to use the global editor

**Description:**
Specifies options that the global editor should use. If this entry does not exist or is empty, Maker uses “%f.” Maker constructs the command line to launch the editor by taking the Editor_Exe content, appending a space to it, and then adding the Editor_Opts entry. Maker cannot modify this entry.

**Example:**

```
[Editor]
editor_name=WinEdit
editor_exe=C:\Winedit\WinEdit.exe
editor_opts=%f
```
Example (MCUTOOLS.INI)

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

```
[Installation]
Path=c:\freescale
Group=Make
[Editor]
editor_name=WinEdit
editor_exe=C:\Winedit\WinEdit.exe
editor_opts=%f
[Options]
DefaultDir=c:\myprj
[Maker]
SaveOnExit=1
SaveAppearance=1
SaveEditor=1
SaveOptions=1
RecentProject0=c:\myprj\project.ini
RecentProject1=c:\otherprj\project.ini
```

Local Configuration File (Usually project.ini)

Maker only reads the default.env file content and does not change that content in any way. Instead, the configuration file stores all configuration properties. Different applications can use the same configuration file.

The shell uses the configuration file with the name “project.ini” in the current directory only, which is why you should also use this name with Maker. Maker can use the editor configuration written and maintained by the shell only when that shell uses the same file as Maker. Apart from this distinction, Maker can use any file name for the project file. The configuration file has the same format as Windows ini files. Maker stores its own entries with the same section.

The current directory is always the directory where the configuration resides. If Maker loads a configuration file from a different directory, then the current directory also changes. When the current directory changes, Maker also re-loads the whole default.env file.
[Editor] Section

**Entry:**

Editor_Name

**Arguments:**

The name of the local editor

**Description:**

Specifies the name displayed in the local editor. This entry has only a descriptive effect. Its content does not apply to starting the editor.

**Saved:**

Maker cannot modify this entry. This entry has the same format as the global editor configuration in the mcutools.ini file.

**Entry:**

Editor_Exe

**Arguments:**

The name of the executable file of the local editor

**Description:**

Specifies file name 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 exists and is not empty.

**Saved:**

Maker cannot modify this entry.

This entry has the same format as the global editor configuration in the mcutools.ini file.

**Entry:**

Editor_Opts

**Arguments:**

The options to use the local editor
**Maker Environment Variables**

*Setting Parameters*

**Description:**
Specifies options that the local editor should use. If this entry does not exist or is empty, Maker uses "%f." Maker constructs the command line to launch the editor by taking the Editor_Exe content, appending a space to it, and then adding the Editor_Opts entry.

**Saved:**
Maker cannot modify this entry. This entry has the same format as the global editor configuration in the mcutools.ini file.

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

**Example (project.ini)**
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

[Maker]
StatusBarEnabled=1
ToolBarEnabled=1
WindowPos=0,1,-1,-1,-1,390,1103,643
WindowFont=-16,500,0,Courier
TipFilePos=0
ShowTipOfDay=1
EditorType=3
RecentCommandLine0=mkall.mak
RecentCommandLine1=cpplib.mak -D(LIBNAME=cpplib)
CurrentCommandLine=mkall.mak
EditorDDEClientName=[open(%f)]
EditorDDET optimistic=system
EditorDDEServiceName=maddev
EditorCommandLine=C:\WinEdit\WinEdit.exe %f /#:%l
```

636  
HC(S)08/RS08 and HC(S)12 Build Tools Utilities Manual
Paths

Most environment variables contain path lists that specify file locations. A path list is a list of directory names separated by semicolons. The path list follows this syntax:

PathList = DirSpec {";" DirSpec).
DirSpec  = ["*"]DirectoryName.

Example:

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

If an asterisk (*) precedes a directory name, the programs recursively searches that whole directory tree for a file, not just the given directory itself. Maker searches directories in the order given in the path list.

Example:

LIBPATH=*C:\INSTALL\LIB

NOTE  Some DOS/UNIX environment variables (like GENPATH, LIBPATH, etc.) are used.

Line Continuation

You can use the line continuation character `\` to specify an environment variable in a environment file (default.env/.hidefaults) over different lines:

FLAGS=\ 
-W2 \ 
-Wpd

These statements are equivalent to FLAGS=-W2 -Wpd

However, using this feature together with paths, such as:

GENPATH=.
TEXTFILE=.
txt
results in GENPATH= .TEXTFILE= .txt

To avoid such problems, use a semicolon ';' at the end of a path if there is a `\` at the end:
GENPATH=.;
TEXTFILE=.;\txt

Input and Output Files

Maker searches for makefiles and include files first in the current directory and then in the GENPATH: #include “File” Path directory.

Maker calls the tools that produce the output files of a make run (except error reports). Refer to the corresponding manuals for the tools you use.

Error Listing

If Maker detects any errors, it creates an error listing file “ERR.TXT”. Maker generates this file in the working directory.

If you start Maker from WinEdit (with ‘%f’ on the command line) or Codewright (with ‘%b%e’ on the command line), it does not produce this error file. Instead, Maker writes the error messages in a special format in a file called EDOUT using the default Microsoft format. Use WinEdit’s Next Error or Codewright’s Find Next Error command to see both the error positions and the error messages.

NOTE  Maker cannot report error-position information with the same precision as a compiler because most of the errors have a long history. Maker can only report the general position, not the position where the error occurred. Most of Maker’s messages lack position information (pos = 0).

Interactive Mode (Main Window Opened)

If you set ERRORFILE: Error File Name Specification, Maker creates a message file with the name specified in this environment variable.

If you do not set ERRORFILE, Maker generates a default file named ERR.TXT in the current directory.

Batch Mode (Main Window Not Opened)

If you set ERRORFILE: Error File Name Specification, Maker creates a message file with the name specified in this environment variable.

If you do not set ERRORFILE, Maker generates a default file named EDOUT in the current directory.
List of Environment Variables

The remainder of this section explains each of the environment variables available to Maker. This section lists options in alphabetical order, with one option to each page.

COMP: Modula-2 Compiler

Tools:
  Maker

Synonym:
  none.

Syntax:
  COMP = <compiler>.

Arguments:
  <compiler>: Used Modula-2 compiler.

Default:
  None.

Description:
  Use this environment variable to specify the Modula-2 compiler.

Example:
  COMP=C:\INSTALL\PROG\TPM.EXE
DEFAULTDIR: Default Current Directory

Tools:
   Compiler, Assembler, Linker, Decoder, Debugger, Libmaker, Maker

Synonym:
   none

Syntax:
   DEFAULTDIR = <directory>.

Arguments:
   <directory>: the default current directory.

Default:
   None.

Description:
   Use this environment variable to specify the default directory for all tools. All the tools indicated above take the specified directory as their current directory, instead of the one defined by the operating system or launching tool (such as the editor).

   This variable is a system-level (global) environment variable. You cannot specify it in a default environment file (DEFAULT.ENV/hidefaults).

Example:
   DEFAULTDIR=C:\INSTALL\PROJECT
ENVIRONMENT: Environment File Specification

Tools:
Compiler, Linker, Decoder, Debugger, Libmaker, Maker

Synonym:
HIENVIRONMENT

Syntax:
ENVIRONMENT = <file>.

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

Default:
None.

Description:
You must specify this variable at system level. The Tool looks in the DEFAULTTDIR: Default Current Directory for a environment file named default.env (.hidefaults on UNIX). Using ENVIRONMENT (set in the autoexec.bat (DOS) or .cshrc (UNIX) file), you can specify a different file name.

This variable is a system-level (global) environment variable. You cannot specify it in a default environment file (DEFAULT.ENV/.hidefaults).

Example:
ENVIRONMENT=\freescale\prog\global.env
ERRORFILE: Error File Name Specification

Tools:

Compiler, Assembler, Maker (restricted)

Synonym:

None

Syntax:

ERRORFILE = <file name>.

Arguments:

<file name>: File name with possible format specifiers.

Description:

The environment variable ERRORFILE specifies the name for the error file that the Tools use. The Compiler and assembler allow substitution in the definition. Possible format specifiers are:

'\%n': Substitute the file name, without the path.

'\%p': Substitute the path of the source file.

'\%f': Substitute the full file name, such as, with the path and name (the same as '%p%n').

In case of an invalid error file name, a notification box appears.

NOTE: Maker does not recognize error files of other tools containing '%' substitutions. Maker reads the string assigned to the environment variable ERRORFILE as filename string without substitutions. So tools that use '%' substitutions for their error output report their error to Maker as the unspecified error message “M5108 called application detected an error”.

Example:

ERRORFILE=MyErrors.err
writes all errors to the file MyErrors.err in the current directory.

ERRORFILE=\tmp\errors
writes all errors to the file errors in the directory \tmp.
ERRORFILE=\%f.err
writes all errors to a file with the same name as the source file, but with extension .err, in the same directory as the source file. For example, if we run a make file \sources\test.mak, Maker generates an error list file \sources\test.err.
ERRORFILE=\dir1\%n.err
For a source file test.mak, Maker generates an error list file \dir1\test.err.
ERRORFILE=\%p\errors.txt
For a source file \dir1\dir2\test.mak, Maker generates an error list file \dir1\dir2\errors.txt.
If you do not set the environment variable ERRORFILE, Maker writes the errors to the file EDOUT in the current directory.

FLAGS: Options for Modula-2 Compiler

Tools:
Maker for Modula-2

Syntax:
FLAGS = {<optionlist>}

Arguments:
<optionlist>: List of options.

Default:
None

Description:
Maker, fed with a Modula-2 main module, starts the compiler with the options specified with FLAGS. The environment variable COMP specifies the Modula-2 compiler.
GENPATH: #include “File” Path

Tools:

Compiler, Linker, Decoder, Debugger, Maker

Synonym:

HIPATH

Syntax:

GENPATH = {<path>}.  

Arguments:

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

Default:

Current directory

Description:

If Maker includes a file from a make file, it searches first in the current directory, then in the directories given by GENPATH.

NOTE

If a directory specification in this environment variables starts with an asterisk (“*”), Maker searches the whole directory tree recursively depth first, that is, it searches all subdirectories and their subdirectories and so on. Within one level in the tree, search order of the subdirectories is indeterminate (these are invalid for Win32).

Example:

GENPATH=\sources\include;..\..\headers;\usr\local\lib
**LINK: Linker for Modula-2**

**Tools:**
Maker for Modula-2

**Syntax:**
\[ \text{LINK} = \{\text{<linker>}\} . \]

**Arguments:**
\[ \text{<linker>: Linker for Modula-2.} \]

**Default:**
none

**Description:**
Maker, fed with a Modula-2 main module, starts the linker specified in this environment variable.

---

**TEXTFAMILY: Text Font Family**

**Tools:**
Compiler (v5.0.x), Assembler (v5.0.x), Linker, Decoder, Libmaker, Maker (v5.0.x)

**Synonym:**
HITEXTFAMILY

**Syntax:**
\[ \text{TEXTFAMILY} = \text{<FontName>} . \]

**Arguments:**
\[ \text{<FontName>: Font family name to use.} \]

**Default:**
Terminal
**TEXTFAMILY: Text Font Family**

**Description:**
Defines the font family to use. The default font family is “Terminal.”

**Example:**

```
TEXTFAMILY=Times
```

**TEXTKIND: Text Font Character Set**

**Tools:**

Compiler (v5.0.x), Assembler (v5.0.x), Linker, Decoder, Libmaker, Maker (v5.0.x)

**Synonym:**
HITEXTKIND

**Syntax:**

```
TEXTKIND = ( OEM | ANSI ).
```

**Arguments:**

- **OEM**: Use OEM font character set.
- **ANSI**: Use ANSI font character set.

**Default:**

**OEM**

**Description:**

Gives the character set, OEM or ANSI. OEM is the default value.

**Example:**

```
TEXTKIND=ANSI
```
TEXTSIZE: Text Font Size

Tools:
Compiler (v5.0.x), Assembler (v5.0.x), Linker, Decoder, Libmaker, Decoder,
Maker (v5.0.x)

Synonym:
HITEXTSIZE

Syntax:
TEXTSIZE = <number>.

Arguments:
<number>: Font size to use.

Default:
14

Description:
Defines the size of the font. The default size is 14 point.

Example:
TEXTSIZE=12
TEXTSTYLE: Text Font Style

Tools:
Compiler (v5.0.x), Assembler (v5.0.x), Linker, Decoder, Libmaker, Maker (v5.0.x)

Synonym:
HITEXTSTYLE

Syntax:
TEXTSTYLE = ( NORMAL | BOLD ) .

Arguments:
NORMAL: Use normal font style (not bold or italic).
BOLD: Use bold font style.

Default:
NORMAL

Description:
Defines the font style to use, NORMAL or BOLD. The default value is NORMAL.

Example:
TEXTSTYLE=BOLD
Building Libraries

This chapter explains how to use the Maker utility to adapt or build your own libraries. Figures and listings in this chapter have the `<target>` identifier instead of a specific CPU name. `<target>` stands for your own target name.

Click any of the following links to jump to the corresponding section of this chapter:

- **Maker Directory Structure**
- **Configuring WinEdit for the Maker**
- **Configuring default.env for the Maker**
- **Building Libraries with Defined Memory Model Options**
- **Building Libraries With Objects Added**
- **Structured Makefiles for Libraries**

Maker Directory Structure

The make files distributed for building the libraries expect the directory structure recommended in the Tools installation. You should have the following items installed in the `c:\freescale` directory.

- **FREESCALE** program folder. Tools .EXE files installed here:
  `c:\freescale\prog`
- Your working directory for building libraries, makefiles, project files, and configuration files installed here:
  `c:\freescale\lib\<target>`
- Binary tool path, defined as a relative path from your working directory in the environment variable `OBJPATH`. Object files and libraries build here:
  `c:\freescale\lib\<target>\lib`
- The `lib` directory contains the library in the preferred object-file format. For targets supporting different object-file formats, other formats reside in these directories (which exist only if the format supports libraries and is not the default):
  - **FREESCALE**: `c:\freescale\lib\<target>\lib.hix`
  - **ELF/DWARF 1.1**: `c:\freescale\lib\<target>\lib.e11`
  - **ELF/DWARF 2.0**: `c:\freescale\lib\<target>\lib.e20`
Building Libraries

Configuring WinEdit for the Maker

- Source paths of the Compiler or Assembler used, defined as a relative path from your working directory in the environment variable GENPATH
  
c:\freescale\lib\<target>\src

- Include path of the Compiler or Assembler used, defined as a relative path from your working directory in the environment variable LIBPATH
  
c:\freescale\lib\<target>\include

Configuring WinEdit for the Maker

Configure WinEdit as follows:
1. Open the Dialog “Project => Configure” in WinEdit.
   This dialog appears only when you open a source file.
2. Load a prepared configuration file with Open or edit the tool definition and save the configuration file.
3. For the Maker configuration (and also the other tools used directly from WinEdit) you must enter the full path to the application in the corresponding text box.
4. Enter the path to your make files in the working directory field.
   Figure 33.1 shows a sample configuration in the Project Management dialog box.

Figure 33.1 Project Management Dialog Box

![Project Management Dialog Box](image)
Configuring `default.env` for the Maker

This section contains a sample "`default.env` (ENVIRONMENT: Environment File Specification) with Maker settings. For building libraries, you need COMP for the compiler, MAKE for the make tool, and LIBM for the library. Additionally, you must specify path environment variables such as OBJPATH and GENPATH. The makefiles introduced in this section also reference these paths.

```
OBJPATH=\.\lib
GENPATH=\.\src
LIBPATH=\.\include
MAKE=\.\.\.\prog\maker.exe
COMP=\.\.\.\prog\c<target>.exe
LIBM=\.\.\.\prog\libmaker.exe
```

Building Libraries with Defined Memory Model Options

Modify memory-model options of a library to build or to extend the built libraries with a new one as follows:

1. Open the file `mkall.mak`.
   
   This file is the main makefile for building libraries. For every library, you specify a command line under the top target `makeall`. An example is:
   
   ```
   $(MAKE) mklib.mak -D(MM=$(FLAGS) -Ms) \
   -D(LIBDIR=$(LIBDIR))\n   -D(LIBNAME=testlib)\n   -D(INCLIBS=ansilib.lib cpplib.lib)
   ```

2. With the command line macro `MM`, specify the options for your library (memory model option and others).
   
   To change the memory model from small to banked, replace `-Ms` in the macro definition with `-Mb`.

**NOTE** The macro definition introduced here is in `-D Define a Macro`. You can specify more than one option switch inside the braces, as in this example:

```
-DM(MM=$(FLAGS) -Ms -Cf)
```
Building Libraries

Building Libraries With Objects Added

3. Specify the library directory in LIBDIR.
   This step is necessary only when you use the default directory \lib, as with processors supporting ELF and Freescale (former Hiware) object-file format.

4. In LIBNAME, name the library to build without an extension. For example, use testlib if the name of the library to build is testlib.lib.

5. Call Maker with mkall.mak.
   The library built with this example includes the ANSI library and the C++ library.

Building Libraries With Objects Added

Add your own objects to a library or build a new one as follows:

1. Copy the ansilib.mak makefile to a makefile with the name of the library you want to build. For example, use mylib.mak if the name of the library you want to build is mylib.lib.

2. Put this makefile in the same directory as the other makefiles.

   NOTE The name of the sublibrary of a built library must be the same as the underlying makefile, with the .lib extension instead of .mak.

3. Remove all object files listed in the macro OBJECTS in mylib.mak.
   If you now list the new makefile “mylib.mak,” you get:

   OBJECTS =
   makeLib: createLib $(OBJECTS)
     echo --- Sublibrary ansilib created
   createLib:
     $(CC) string.c assert.c
     $(LIBM) string.o + assert.o = $(OBJPATH)$(LIBNAME).lib
     del $(OBJPATH)string.o
     del $(OBJPATH)assert.o
   .c.o:
     $(CC) *.c
     $(LIBM) $(OBJPATH)$(LIBNAME).lib+*.o = $(OBJPATH)$(LIBNAME).lib
     del $(OBJPATH)/*.o
4. List your object files with the .o extension in the OBJECTS macro.
   Place your library source files in the folder specified in GENPATH: #include “File” Path.

5. Open the mkall.mak file.

   mkall.mak is the main makefile for building libraries. For every library, you specify a command line under the top target makeall:

   An example is:
   ```bash
   $(MAKE) mklib.mak -D(MM=$(FLAGS) -Ms) \ 
   -D(LIBNAME=testlib)\ 
   -D(STARTANSIOBJ=start<target>s)\ 
   -D(STARTCPPOBJ=strt<target>sp)\ 
   -D(INCLIBS=mylib.lib)
   ```

6. In the passed INCLIBS command-line macro, specify the sublibrary names.

   In the example above, Maker builds only the sublibrary mylib.lib with mylib.mak. In this example, we list only one sublibrary. You can add additional sublibraries to the list, separated by spaces.

7. In LIBNAME, specify the name of the built library without the extension.

   The other macros passed specify the startup files to build. Maker does not insert the startup files into the library but instead builds them separately.

**NOTE** The name of the library to build, specified in LIBNAME, must be different from the name of the sublibrary included, such as mylib in the example. If not, Maker deletes the built library just after building it. (Maker deletes the sublibrary after adding it to the built library.)

8. Call Maker with mkall.mak

   Your library builds among the others.
Structured Makefiles for Libraries

Building a library works on three makefile levels, as shown in Figure 33.2.

Figure 33.2 Building a Library

Top Make File
mkall.mak

Build Library Make File
mklib.mak

Customized Library Makefile
example: ansilib.mak

Customized Library Makefile
example: cpplib.mak

This layering is comparable to the modular concept of procedural programming languages. An upper makefile calls Maker with the makefile and the arguments passed over command-line macros. The top layer makefile mkall.mak, for example, calls the makefile mklib.mak to build one library and passes the memory model, the name of the library to build, the name of the participant sub libraries, and the startup files build.

A sample makefile, mkall.mak, looks like this:

```bash
FLAGS   =  ## insert here the global options for all libraries
makeall:
  -dosprmt.pif /c del lib\*.*
  echo --- Making all libraries:
$(MAKE) mklib.mak -D(MM=$(FLAGS) -Ms) -D(LIBNAME=ansis) \ 
  -D(STARTANSIOBJ=start<target>s) \ 
  -D(STARTCPPOBJ=strt<target>sp) \ 
  -D(INCLIBS=ansilib.lib cpplib.lib)
$(MAKE) mklib.mak -D(MM=$(FLAGS) -Ms -Cf) \ 
  -D(LIBNAME=ansisf) \ 
  -D(STARTANSIOBJ=start<target>s) \ 
  -D(STARTCPPOBJ=strt<target>sp) \ 
  -D(INCLIBS=ansilib.lib cpplib.lib)
$(MAKE) mklib.mak -D(MM=$(FLAGS) -Mb) -D(LIBNAME=ansib)  
```

654 HC(S)08/RS08 and HC(S)12 Build Tools Utilities Manual
Building Libraries

Structured Makefiles for Libraries

The first command for the top target `makeall` deletes all libraries and object files previously built.

One Maker call with `$ (MAKE)` evaluates Maker over the environment variable `MAKE` in `default.env`, which corresponds to building one library.

- The first Maker call of `mklib.mak`, for example, builds an ANSI library for the small memory model (with option `-Ms` passed over the command-line macro `MM`).
- `mklib.mak` expects these command-line macros
  - `MM` = options for the memory model,
  - `LIBNAME` = name of the produced library
  - `STARTUP` = name of the ANSI-C Startup file
  - `STARTCPP` = name of the C++ Startup file
  - `INCLIBS` = in library of included sub libraries

In the example, we pass the library names `cpplib.lib` and `ansilib.lib` in the `INCLIBS` command-line macro. The `mklib.mak` makefile appears below:

```
NOTE
Do not modify `mklib.mak`. Instead, use `mkall.mak` to specify the compiler options, the sublibrary list, and your own sublibraries, such as `ansilib.mak`, `cppplib.mak`, and the example, `mylib.mak`.
```

```
CC = $(COMP) $(MM)
makeall: startup createLib $(INCLIBS)
echo "--- all done! ---"
startup: start<target>.c
echo "--- making startup"
$(CC) $(GENPATH)/start<target>.c
copy $(OBJPATH)/start<target>.o
$(OBJPATH)/$(STARTANSIOBJ).o
$(CC) -C++ $(GENPATH)/start<target>.c
copy $(OBJPATH)/start<target>.o
$(OBJPATH)/$(STARTCPPOBJ).o
del $(OBJPATH)/start<target>.o
```

Building Libraries
Structured Makefiles for Libraries

```
createLib:
    echo "--- startup done"
    echo "--- creating library"
    $(LIBM) $(OBJPATH)$(STARTANSIOBJ).o =
    $(OBJPATH)$(LIBNAME).lib
    $(LIBM) $(OBJPATH)$(LIBNAME).lib -
    $(OBJPATH)$(STARTANSIOBJ).o =\n    $(OBJPATH)$(LIBNAME).lib
    $(LIBM) $(OBJPATH)$(LIBNAME).lib ?
    $(OBJPATH)$(LIBNAME).lst
    echo "--- library done"
    .mak.lib:
    echo "--- making and add $* library"
    $(MAKE) $*.mak -D(CC=$(CC)) -D(LIBNAME=$*)
    $(LIBM) $(OBJPATH)$(LIBNAME).lib + $(OBJPATH)$*.lib =\n    $(OBJPATH)$(LIBNAME).lib
    del $(OBJPATH)$*.lib
    del $(OBJPATH)$*.lst
```

The makefile uses build rules. For each built library, the makefile mylib.mak must reside in the working directory. The makefile collects a group of object files. Maker calls the makefile, passing these command-line arguments as parameters:

- **CC** = compiler with option list
- **LIBNAME** = name of the produced library

These settings depend on settings already passed from mkall.mak. The sublibraries built with the delivered makefiles are ansilib.mak and cpplib.mak.
Maker Options

Maker offers some options that control its process. Options have a minus-dash (‘-’) followed by one or more letters or digits. Anything not starting with a dash/minus represents the name of a make file to process. You can specify Maker options on the command line or interactively in the “Advanced Option Settings” dialog.

Click any of the following links to jump to the corresponding section of this chapter:

- Option Groups
- Option Details

Option Groups

The following table groups Maker options. These groups correspond to tabs in the Options Settings dialog.

Table 34.1 Maker Option Groupings

<table>
<thead>
<tr>
<th>Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTPUT</td>
<td>Specification of command execution and output print</td>
</tr>
<tr>
<td>INPUT</td>
<td>Specification of command-line handling, such as macro definitions and unknown-macro expansions.</td>
</tr>
<tr>
<td>MESSAGES</td>
<td>Message handling, such as specification of format, kind, and number of Maker printed messages</td>
</tr>
<tr>
<td>MODULA-2</td>
<td>Modula-2 make-specific options. (No effect for C-users)</td>
</tr>
<tr>
<td>VARIOUS</td>
<td>Does not appear in the dialog box</td>
</tr>
<tr>
<td>HOST</td>
<td>Host-related options</td>
</tr>
</tbody>
</table>
Option Details

This section describes each Maker option. The following table lists each option according to topic.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td>Output, Input, Messages, Modula-2</td>
</tr>
<tr>
<td>Syntax</td>
<td>Specifies the syntax of the option in an EBNF format.</td>
</tr>
<tr>
<td>Arguments</td>
<td>Describes and lists optional and required arguments.</td>
</tr>
<tr>
<td>Default</td>
<td>Shows the default option setting.</td>
</tr>
<tr>
<td>Description</td>
<td>Describes the option and its use.</td>
</tr>
<tr>
<td>Example</td>
<td>Gives a sample usage, effects, Maker settings, and source code where applicable. The examples appear as an entry in the default.env file for PC or in the .hidefaults file for UNIX.</td>
</tr>
</tbody>
</table>

-A: Warning for Missing .DEF File

Group:
MODULA-2

Syntax:
-A

Arguments:
None

Description:
Invokes a warning for a missing .DEF file and affects only the processing of Modula-2 makefiles.

Example:
```
maker test.mod -M -A
```
-C: Ignore Case

Group:
   INPUT

Syntax:
   \(-C\)

Arguments:
   None.

Description:
The make utility has default case sensitivity. Use this option to disable case sensitivity and treat lowercase characters the same as uppercase characters.

Example:
   
   \texttt{maker test.mak \ -o}
   
   In the file \texttt{test.mak}:
   
   \texttt{OBJECTFILES = startup.o fibo.o}
   
   \texttt{makeAll: \ $(ObjectFiles)}
   
   This line with \texttt{-c} is equivalent to:
   
   \texttt{makeAll: \ $(OBJECTFILES)}

-D: Define a Macro

Group:
   INPUT

Syntax:
   \(-D \ <\text{macroname}\> \ = \ <\text{value}\>\)

Arguments:
   The macro definition string "\(<\text{macroname}\> \ = \ <\text{value}\>\).
**Description:**
This option defines command-line macros. Command-line macros define macros and arguments for the make file. A macro defined this way has a higher priority than a macro defined in the makefile. Because you separate the arguments in the command line with spaces, you cannot place spaces in a command-line macro.

**Examples:**
- \(-d\text{COMP=chc12.exe}\)
- \(-d\text{COMP=chc12.exe} \ -\text{Li} \ -\text{Wi}\)
- \(-d\{\text{MAKE=}\text{Maker.exe} \ -s \ -d(\text{COMP=}$(\text{COMP}))\}\)
Syntax:
   -E

Arguments:
   None

Description:
This macro discards errors for unknown macros referenced in the makefile. Maker
substitutes an unknown macro with an empty string.

Example:
    maker -m test.mod -e

-Env: Set Environment Variable

Group:
    HOST

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

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

Description:
This option sets an environment variable.

Example:
    Maker -Env GENPATH=\myfiles
    This line is the same as:
    \GENPATH=\myfiles
    in the default.env file.

NOTE: Called applications do not inherit this option. Unless these tools have the same
option, they may not find the same files as Maker.
-H: Short Help

Group:
  VARIOUS

Syntax:
  -H

Arguments:
  None

Description:
  Prints a list of available options

-I: Ignore Exit Codes

Group:
  OUTPUT

Syntax:
  -I

Arguments:
  None

Description:
  This option lets Maker ignore exit codes of the called programs. Maker continues
  processing even if the called application reports a fatal error or creation of the
  corresponding process fails. Use this option for testing purposes, where Maker resolves
  only the dependencies of a make file.

Example:
  maker -m test.mod -i
-L: List Modules

Group:
MODULA-2

Syntax:
-L <listfile>

Arguments:
File name of the generated listing file

Description:
This option lists compiled files in build order in the file specified in the argument <listfile>. This option affects only the processing of Modula-2 makefiles.

Example:
maker -m test.mod -l test.lst

-Lic: License Information

Group:
VARIOUS

Syntax:
-Lic

Arguments:
None.

Description:
This option prints the current license information (demo or full version). This information also appears in the About box.

Example:
maker -Lic
-LicA: License Information About Every Feature in Directory

Group:
  VARIOUS

Syntax:
  -LicA

Arguments:
  None

Description:
This option prints license information for every tool or DLL in the directory were the executable resides (demo or full tool version). Because this option must analyze every single file in the directory, it may take a long time.

Example:
  maker -LicA

-LicBorrow: Borrow License Feature

Group:
  HOST

Syntax:
  -LicBorrow <feature>";"<version>";"<Date>

Arguments:
  <feature>: The feature name to be borrowed (e.g. HI100100).
  <version>: Optional version of the feature to be borrowed (e.g. 3.000).
  <date>: Date with optional time until when the feature shall be borrowed (e.g. 15-Mar-2004:18:35).
**Maker Options**

**Option Details**

**Description:**

This option allows you to borrow a license feature until a given date/time. Borrowing allows you to use a floating license even if disconnected from the floating license server.

You need to specify the feature name and the date until you want to borrow the feature. If the feature you want to borrow is a feature belonging to the tool where you use this option, then you do not need to specify the version of the feature (because the tool knows the version). However, if you want to borrow any feature, you need to specify as well the feature version of it.

You can check the status of currently borrowed features in the tool about box.

You only can borrow features, if you have a floating license and if your floating license is enabled for borrowing. See as well the provided FLEXlm documentation about details on borrowing.

**Example:**

```
maker -LicBorrowHI100100;3.000:12-Mar-2004:18:25
```

**-LicWait: Wait Until Floating License Is Available from Floating License Server**

**Group:**

HOST

**Syntax:**

```
-LicWait
```

**Arguments:**

None

**Description:**

By default, if a license is not available from the floating license server, then the application will immediately return. With -LicWait set, the application will wait (blocking) until a license is available from the floating license server.

**Example:**

```
maker -LicWait demo.mak
```
Maker Options
Option Details

-M: Produce Make File

Group:
MODULA-2

Syntax:
-M [ <makefile> ]

Arguments:
File name of the generated makefile

Description:
This option generates a makefile. If this option immediately follows a file name, Maker writes the makefile to that file; otherwise, the makefile has the same name as the main module, but with suffix .MAK. This makefile uses macros by referencing above environment variables.

Example:
maker test.mod -m test.mak

-MkAll: Make Always

Group:
INPUT

Syntax:
-MkAll

Arguments:
None

Description:
This option skips Maker time-checking. Maker rebuilds up-to-date files. Use this option for updating the application after a change not covered by makefile dependencies.
Example:

    maker test.mak -mkall

-N: Display Notify Box

Group:
    MESSAGE

Syntax:
    -N

Arguments:
    None

Description:
Maker displays an alert box if an error occurred during the make process. Use this option for error checking, since Maker waits for the you to acknowledge the message, suspending makefile processing. (N stands for Notify.)

This option exists only on the IBM PC version of Maker.

Example:

    maker test.mak -N

    If an error occurs during the make process, Maker opens a dialog box.

-NoBeep: No Beep in Case of an Error

Group:
    MESSAGE

Syntax:
    -NoBeep

Arguments:
    None
**Maker Options**

*Option Details*

---

**Description:**

There is a default beep notification of errors at the end of processing. Use this option to switch off the beep.

**Example:**

```
maker test.mak -NoBeep
```

---

**-NoCapture: Do Not Redirect stdout of Called Processes**

**Group:**

OUTPUT

**Syntax:**

```
-NoCapture
```

**Arguments:**

None

**Description:**

Maker’s default behavior is to redirect from `stdout` the output text of called applications. Use this option to prevent redirection and text output for errors. This option affects only text output, since Maker does not detect the called application issuing the error.

This option accelerates the make process and older applications that do not support output. Using this option is equivalent to placing “*” at the start of every command line.

**Example:**

```
maker test.mak -NoCapture
```

---

**-NoEnv: Do Not Use Environment**

**Group:**

Startup. (You cannot interactively specify this option.)
**Syntax:**
- NoEnv

**Arguments:**
None

**Default:**
None

**Description:**
Use this option only on the command line while starting the application. The application does not use any environment (default.env, project.ini or tips file).

**Example:**
```
maker.exe -NoEnv
```

---

**-O: Compile Only**

**Group:**
MODULA-2

**Syntax:**
- O

**Arguments:**
None.

**Example:**
```
maker test.mod -o
```

**Description:**
Use this macro to have Maker perform only compile steps for a Modula-2 build. Maker does not call the linker. This option affects only Modula-2 makefile processing.
Maker Options
Option Details

-S: Silent Mode

Group:
OUTPUT

Syntax:
-S

Arguments:
None

Example:
maker test.mod -s

Description:
Maker does not echo executed commands. Use this option to examine only Maker
texts or those of the called tools, where an otherwise long list of executed commands
is inconvenient.

-V: Prints the Version

Group:
VARIOUS

Syntax:
-V

Arguments:
None.

Description:
This option prints the version of all Maker components

NOTE Use this option to determine the current Maker directory.
Example:
-v on the command line

-View: Application Standard Occurrence (PC)

Group:
HOST

Syntax:
-View <kind>

Arguments:
<kind> is one of the following:
- Window: Application window with default window size
- Min: Minimized application window
- Max: Maximized application window
- Hidden: Invisible application window

Default:
Application started with arguments: Minimized.
Application started without arguments: Window.

Description:
The linker, compiler, and Maker usually start in a normal window if you specify no arguments. If you start the application with arguments (such as compiling or linking a file), the application runs in a minimized window to allow batch processing.
- With -ViewWindow, the application has its normal window.
- With -ViewMin, the application appears as an icon in the task bar.
- With -ViewMax, the application is maximized.
- With -ViewHidden, the application processes arguments (files to be compiled or linked) in the background.

However, if you are use “-N: Display Notify Box”, a dialog box may appear.

Example:
-ViewMin fibo.mak
**-WErrFile: Create “err.log” Error File**

**Group:**
MESSAGE

**Syntax:**
-WErrFile (On | Off)

**Arguments:**
None

**Default:**
err.log created/deleted

**Description:**
Return codes provide error feedback from the compiler to called tools. In 16-bit Windows environments, return codes are unavailable, so the err.log file contains error messages or else reports no errors. In UNIX or WIN32, return codes are available, so the err.log file is unnecessary. To use the 16-bit Maker with this tool, you must create the error file in order to capture errors.

**Example:**
-WErrFileOn
err.log created/deleted after the application finishes.
-WErrFileOff
existing err.log remains unmodified.

**-Wmsg8x3: Cut File Names in Microsoft Format to 8.3**

**Group:**
MESSAGE

**Syntax:**
-Wmsg8x3
Arguments:
None

Description:
Some editors, such as early versions of WinEdit, expect the file name in the Microsoft message format (a strict 8.3 format), meaning the file name can have at most 8 characters with no more than 3 characters in the extension. Using Win95 or WinNT, longer filenames are possible. Use the -Wmsg8x3 option to have Maker truncate the filename in the Microsoft message to the 8.3 format.

This option is only available on an IBM PC version of Maker and affects only error output.

-WmsgCE: RGB Color for Error Messages

Group:
MESSAGE

Scope:
Function

Syntax:
-WmsgCE <RGB>.

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

Default:
-WmsgCE16711680 (rFF g00 b00, red)

Description:
Use this option to change the error-message color. Specify an RGB (Red-Green-Blue) decimal value. Hexadecimal needs a 0X prefix, (for gray errors: -WmsgCE0x808080).

Example:
-WmsgCE255
changes the error messages to blue.
-WmsgCF: RGB Color for Fatal Messages

Group:
MESSAGE

Scope:
Function

Syntax:
-WmsgCF <RGB>.

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

Default:
-WmsgCF8388608 (r80 g00 b00, dark red)

Description:
Use this option to change the fatal message color. Specify an RGB (Red-Green-Blue) decimal value. Hexadecimal needs a 0X prefix, (for gray errors: -WmsgCF0x808080).

Example:
-WmsgCF255
changes the fatal messages to blue.

-WmsgCI: RGB Color for Information Messages

Group:
MESSAGE

Scope:
Function

Syntax:
-WmsgCI <RGB>.
Arguments:

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

Default:

-WmsgCI32768 (r00 g80 b00, green)

Description:

Use this option to change the information message color. Specify an RGB (Red-Green-Blue) decimal value. Hexadecimal needs a 0X prefix, (for gray errors: -WmsgCI0x808080).

Example:

-WmsgCI255

changes the information messages to blue.

-WmsgCU: RGB Color for User Messages

Group:

MESSAGE

Scope:

Function

Syntax:

-WmsgCU <RGB>.

Arguments:

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

Default:

-WmsgCU0 (r00 g00 b00, black)

Description:

Use this option to change the user message color. Specify an RGB (Red-Green-Blue) decimal value. Hexadecimal needs a 0X prefix, (for gray errors: -WmsgCU0x808080).
Example:

- WmsgCU255
  changes the user messages to blue.

-WmsgCW: RGB Color for Warning Messages

Group:
  MESSAGE

Scope:
  Function

Syntax:

- WmsgCW <RGB>.

Arguments:

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

Default:

- WmsgCW255 (r00 g00 bFF, blue)

Description:

Use this option to change the warning message color. Specify an RGB (Red-Green-Blue) decimal value. Hexadecimal needs a 0X prefix, (for gray errors: -WmsgCW0x808080).

Example:

- WmsgCWO
  changes the warning messages to black.

-WmsgFb: Set Message File Format for Batch Mode

Group:
  MESSAGE
Maker Options
Option Details

Syntax:
   -WmsgFb [v | m]

Arguments:
   v: Verbose format
   m: Microsoft format

Default:
   -WmsgFbm

Description:
You can start Maker with additional arguments. If you start Maker with arguments such as the ‘%f’ argument from WinEdit, Maker processes makefiles in batch mode. In batch mode, Maker windows do not appear, and the job terminates after completion. If a Tool runs in batch mode, messages write to a file instead of to the screen.

Tools use the default Microsoft message format. With -WmsgFb, you can change the default format from the Microsoft format to a more verbose error format with line, column, and source information. This format is not as important for the make tool as for other Tools. Most Maker messages do not contain source-position information.

Maker reads the error output of called tools over the error-output file and prints lines containing the keywords ERROR, FATAL, WARNING, or INFORMATION in the output file.

Example:
   maker test.mak -WmsgFbi

- WmsgFi: Set Message Format for Interactive Mode

Group:
   MESSAGE

Syntax:
   -WmsgFi [v | m]

Arguments:
   v: Verbose format
   m: Microsoft format
**Maker Options**

*Option Details*

### Default:

- `-WmsgFiv`

### Example:

```
maker test.mak -WmsgFim
```

### Description:

Same as `-WmsgFb` for file output. `-WmsgFi` is for the standard output of the interactive Maker. If you start Maker without additional arguments, it operates in interactive mode (with a visible window).

Maker uses the default verbose error file format to write error, warning, and information messages.

With `-WmsgFi`, you can change the default format from verbose (with source, line and column information) to the Microsoft format (only line information).

---

**-WmsgFob: Message Format for Batch Mode**

### Syntax:

```
-WmsgFob<string>.
```

### Arguments:

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

### Default:

```
-WmsgFob"%f%%e(%l): %K %d: %m\n"
```
Description:
This option modifies the default message format in batch mode. Maker supports these formats (assuming the source file is `x:\freescale\mymakefile.makefile`):

```
<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%s</td>
<td>Source Extract</td>
<td>x:\freescale\mymakefile.makefile</td>
</tr>
<tr>
<td>%p</td>
<td>Path</td>
<td>x:\freescale\mymakefile</td>
</tr>
<tr>
<td>%f</td>
<td>Path and name</td>
<td>x:\freescale\mymakefile</td>
</tr>
<tr>
<td>%n</td>
<td>File name</td>
<td>mymakefile</td>
</tr>
<tr>
<td>%e</td>
<td>Extension</td>
<td>.makefile</td>
</tr>
<tr>
<td>%N</td>
<td>File (8 chars)</td>
<td>mymakefi</td>
</tr>
<tr>
<td>%E</td>
<td>Extension (3 chars)</td>
<td>.mak</td>
</tr>
<tr>
<td>%l</td>
<td>Line</td>
<td>3</td>
</tr>
<tr>
<td>%c</td>
<td>Column</td>
<td>47</td>
</tr>
<tr>
<td>%o</td>
<td>Pos</td>
<td>1234</td>
</tr>
<tr>
<td>%K</td>
<td>Uppercase kind</td>
<td>ERROR</td>
</tr>
<tr>
<td>%k</td>
<td>Lowercase kind</td>
<td>error</td>
</tr>
<tr>
<td>%d</td>
<td>Number</td>
<td>M1815</td>
</tr>
<tr>
<td>%m</td>
<td>Message</td>
<td>text</td>
</tr>
<tr>
<td>%%</td>
<td>Percent</td>
<td>%</td>
</tr>
</tbody>
</table>
\nExample:

```

```
COMPOPTIONS=-WmsgFoib%f%e(%l): %k %d: %r
```

produces a message in this format:
```
X:\C.mak(3): information M1234: text
```

-WmsgFoi: Message Format for Interactive Mode

Syntax:
```
-WmsgFoi<string>
```

---

HC(S)08/RS08 and HC(S)12 Build Tools Utilities Manual 679
Maker Options
Option Details

Arguments:

<string>: format string (see below).

Default:

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

Description:
This option modifies the default message format in interactive mode. Maker supports these formats (assuming the source file is x:\freescale\mymakefile.mak):

<table>
<thead>
<tr>
<th>Format Description Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%s     Source Extract</td>
</tr>
<tr>
<td>%p     Path                x:\freescale\</td>
</tr>
<tr>
<td>%f     Path and name       x:\freescale\mymakefile</td>
</tr>
<tr>
<td>%n     File name           mymakefile</td>
</tr>
<tr>
<td>%e     Extension           .makefile</td>
</tr>
<tr>
<td>%N     File (8 chars)      mymakefi</td>
</tr>
<tr>
<td>%E     Extension (3 chars) .mak</td>
</tr>
<tr>
<td>%l     Line                3</td>
</tr>
<tr>
<td>%c     Column              47</td>
</tr>
<tr>
<td>%o     Pos                 1234</td>
</tr>
<tr>
<td>%K     Uppercase kind      ERROR</td>
</tr>
<tr>
<td>%k     Lowercase kind      error</td>
</tr>
<tr>
<td>%d     Number              M1815</td>
</tr>
<tr>
<td>%m     Message              text</td>
</tr>
<tr>
<td>%%     Percent              %</td>
</tr>
<tr>
<td>\n     New line</td>
</tr>
</tbody>
</table>

Example:

COMPOPTIONS=-WmsgFoi"%f%e(%l):%k%d: %m\n"
produces a message in this format:

X:\C.C(3): information M1234: text
-\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 no file information exists for a message, such as when a message does not
   correspond to a specific file. In this case, use this message format.

   Format Description Example
   -------------------------------
   | %K     | Uppercase kind       | ERROR |
   | %k     | Lowercase kind       | error |
   | %d     | Number               | M1815 |
   | %m     | Message              | text  |
   | %     | Percent              | %     |
   | \n     | New line             |       |

Example:
   COMPOPTIONS=-\WmsgFonf"%k %d: %m\n"
   produces a message in this format:
   information M1234: text
**-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 no position information exists for a message, such as when a message does not correspond to a certain position. In this case, use this message format.

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%K</td>
<td>Uppercase kind</td>
<td>ERROR</td>
</tr>
<tr>
<td>%k</td>
<td>Lowercase kind</td>
<td>error</td>
</tr>
<tr>
<td>%d</td>
<td>Number</td>
<td>M1815</td>
</tr>
<tr>
<td>%m</td>
<td>Message</td>
<td>text</td>
</tr>
<tr>
<td>%%</td>
<td>Percent</td>
<td>%</td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
<td></td>
</tr>
</tbody>
</table>

**Example:**

COMPOPTIONS=-WmsgFonf"%k %d: %m\n"

produces a message in this format:

information M1234: text
-WmsgNe: Number of Error Messages

Group:
MESSAGE

Syntax:
-WmsgNe <number>

Arguments:
<number>: Maximum number of error messages

Default:
50

Description:
Sets the number of error messages that Maker generates until it stops processing files.

NOTE Subsequent error messages that depend on previous ones may be confusing.

Example:
maker.exe -WmsgNe2 test.mak
Maker stops processing files after two error messages.

-WmsgNi: Number of Information Messages

Group:
MESSAGE

Syntax:
-WmsgNi<number>

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

50

Description:
Sets the number of information messages

Example:

maker.exe -WmsgNi10 test.mak
Maker logs only ten information messages.

-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:
Maker produces messages in categories other than WARNING, INFORMATION, ERROR, and FATAL. Use this option to disable such messages. The idea is to reduce the number of messages and simplify error parsing of other tools.
a : Use this suboption to disable messages listing all included files.
b : Use this suboption to disable messages about reading files, such as the files used as input.
-WmsgNu: Use this suboption to disable messages about generated files.

-WmsgNw: Use this suboption to disable messages about statistical information, such as code size, RAM/ROM usage, and so on. Maker generates these messages after finishing file processing.

-WmsgNv: Use this suboption to disable informal messages (such as memory model and floating point format).

**NOTE** Depending on the application, not all suboptions may make sense, in which case Maker ignores them.

**Example:**

```
-WmsgNu=c
```

**-WmsgNw: Number of Warning Messages**

**Group:**

MESSAGE

**Syntax:**

```
-WmsgNw<number>
```

**Arguments:**

<number>: Maximum number of warning messages.

**Default:**

50

**Description:**

This option sets the number of warning messages.

**Example:**

```
maker.exe -WmsgNw15 test.mak
```

Maker logs only 15 warning messages.
Maker Options
Option Details

-WmsgSd: Setting a Message to Disable

Group:
MESSAGE

Syntax:
-WmsgSd<number>

Arguments:
<number>: Message number to disable, for example, message 1801

Default:
None

Description:
This option disables a message so that it does not appear in the error output.

Example:
maker.exe test.mak -WmsgSd1801

-WmsgSe: Setting a Message to Error

Group:
MESSAGE

Syntax:
-WmsgSe<number>

Arguments:
<number>: Message number to report as an error, for example, message 1853

Default:
None
**-WmsgSi: Setting a Message to Information**

**Group:**
MESSAGE

**Syntax:**
-`-WmsgSi<number>`

**Arguments:**
<number>: Message number to report as information, for example, message 1853

**Default:**
None

**Description:**
This option changes a message to an information message

**Example:**

```
maker.exe test.mak -WmsgSi1801
```
Arguments:

<number>: Message number to report as a warning, for example, message 2901

Default:

None

Description:

Changes a message to a warning message

Example:

maker.exe test.mak -WmsgSw1801

-WmsgVrb: Verbose Mode

Group:

MESSAGE

Syntax:

-WmsgVrb

Arguments:

None

Default:

None.

Description:

Maker prints the error messages to an error file, as explained in the section Message/Error Feedback

Example:

maker.exe test.mak -WmsgVrb
-WOutFile: Create Error Listing File

Group:
MESSAGE

Syntax:
-WOutFile (On | Off)

Arguments:
None

Default:
Creates error listing file

Description:
Controls whether Maker creates an error-listing file. The error-listing file contains a list of messages and errors generated during compilation. Since Maker can also handle the text-error feedback with pipes to the calling application, you can get this feedback without an explicit file. The ERRORFILE: Error File Name Specification environment variable specifies the name of the listing file.

Example:
-WOutFileOn
Maker generates the error file as specified with ERRORFILE.
-WOutFileOff
Maker does not generate an error file.

-WStdout: Write to Standard Output

Group:
MESSAGE

Syntax:
-WStdout (On | Off)
Maker Options
Option Details

Arguments:
None

Default:
Maker writes output to stdout.

Description:
Controls output if Maker writes the text of an error file to stdout.

Example:
- WStdoutOn
  Maker writes all messages to stdout.
- WErrFileOff
  Maker writes nothing to stdout.

-W1: No Information Messages

Group:
MESSAGE

Syntax:
-W1

Arguments:
None

Default:
None

Description:
This option prevents Maker from printing INFORMATION messages. Only WARNINGs and ERROR messages go to output.

Example:
maker test.mak -W1
-W2: No Information and Warning Messages

**Group:**
MESSAGE

**Syntax:**
-W2

**Arguments:**
None.

**Description:**
This option suppresses all INFORMATION messages; prints only WARNING and ERROR messages (see Kinds of Maker Messages).

**Example:**
```
  maker test.mak -W2
```
Maker Messages

This chapter describes Maker printed messages. Maker itself does not issue most of the messages generated in a make process, they usually result from other processes or programs started from Maker. Maker reads the output of the other tools, allowing it to classify messages that the called application issues by message type. For example, you can reclassify ERROR messages from called programs as Maker warnings.

If Maker prints out a message, the message contains a message code (‘M’ for Maker) and a four- to five-digit number. Use this number to search a message quickly. This section documents in ascending order all messages that the makertool generates.

Each message also has a description and sometimes a short example with a possible solution or tip to fix a problem. Each message notes a particular type, such as ERROR.

Click any of the following links to jump to the corresponding section of this chapter:

- Kinds of Maker Messages
- Makefile Messages
- Exec Process Messages
- Modula-2 Maker Messages

Kinds of Maker Messages

The following table lists and describes the five kinds of Maker messages.

<table>
<thead>
<tr>
<th>Message</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information</td>
<td>A message prints and the make process continues.</td>
</tr>
<tr>
<td>Warning</td>
<td>A message prints and the make process continues.</td>
</tr>
<tr>
<td>Error</td>
<td>A message prints and the make process stops.</td>
</tr>
<tr>
<td>Fatal</td>
<td>A message prints and the make process aborts.</td>
</tr>
<tr>
<td>Disable</td>
<td>A message gets disabled and its output suppressed.</td>
</tr>
</tbody>
</table>
Makefile Messages

This section lists and describes error messages that can appear when:

- Maker detects an error in the makefile
- A called application detects an error that Maker catches

M1: Unknown Message Occurred

Message Type
[FATAL]

Description:
Maker tried to send an undefined message. This internal error should not occur. Please report it to your distributor.

M2: Message Overflow, Skipping <kind> Messages

Message Type
[INFORMATION]

Description:
The compiler showed the number of messages of the specific kind as controlled with the options `-WmsgNi: Number of Information Messages`, `-WmsgNw: Number of Warning Messages` and `-WmsgNe: Number of Error Messages`. Maker does not display further options of this kind.

TIP Use the options WmsgNi, WmsgNw and WmsgNe to change the number of messages.
M50: Input File ‘<file>’ Not Found

Message Type
[FATAL]

Description:
The Application did not find a file needed for processing.

TIP Make sure the file really exists and that you are using a file name containing spaces (in this case, you have to quote it).

M51: Cannot Open Statistic Log File <file>

Message Type
[WARNING]

Description:
Maker could not open a statistic output file, therefore it generated no statistics.

NOTE If a tool does not support statistical log files, the message still exists but Maker does not issue it.
M64: Line Continuation Occurred in <FileName>

Message Type
[INFORMATION]

Description:
In any environment file, the backslash character (\) at the end of a line denotes a line continuation. Maker handles this line and the next one as a single line. Because the backslash is also the path-separation character in MS-DOS, paths often incorrectly end in \\'. Instead, use a period (.) after the last backslash unless you really want a line continuation.

Example:
Current Default.env:
...
LIBPATH=c:\freescale\lib\nOBJPATH=c:\freescale\work
...
which Maker interprets as:
...
LIBPATH=c:\freescale\libOBJPATH=c:\freescale\work
...

TIP    Append a period (.) behind the backslash (\).

...(continued)


NOTE    Because this information occurs during the compiler’s initialization phase, the
         ‘C’ may not occur in the error message but may appear as “64: Line
         Continuation occurred in <FileName>”.
M65: Environment Macro Expansion Error 'description' for <vari-ablename>

Message Type
[INFORMATION]

Description:
A problem occurred during an environment-variable macro substitution. Possible causes are that the named macro did not exist or some length limitation occurred. Recursive macros may also cause this message.

Example:
Current Default.env:
...
LIBPATH=${LIBPATH}
...

TIP Check the definition of the environment variable.

M66: Search Path <Name> Does Not Exist

Message Type
[INFORMATION]

Description:
The tool could not find a file. During the failed search, the tool encountered a nonexistent path.

TIP Check the spelling of your paths.
Update the paths when moving a project.
Use relative paths in your environment variables.
Make sure network drives are available.
M5000: User Requested Stop

**Message Type**

[ERROR]

**Description:**
The user clicks the **Stops the current make process** icon. A message dialog prompts you to continue or interrupt the current make process.

M5001: Error in Command Line

**Message Type**

[ERROR]

**Description:**
Maker detected a syntax error in the command line. Maker scans only the tokens that start with a dash (-) (which signals options) but leaves the other names in command line unscanned. Because Maker assumes that these tokens represent filenames, it answers only option errors with this message. It prints M5019 for other syntactical errors.

**Example:**

```
maker -Y
## Y is an illegal option
```

**TIP**

Call Maker with the -h argument for a list of options.
M5002: Can't Return to `<makefile>` at End of Include File

**Message Type**

[ERROR]

**Description:**
The makefile executed before opening the include file and Maker cannot reopen it again.

**TIP**  Make sure the makefile exists.

M5003: Illegal Dependency

**Message Type**

[ERROR]

**Description:**
Only identifiers or filenames can reside in the dependency list. Maker reports other tokens as invalid.

**Example:**
```
makeall:
    inout.o message.o main.o (*)
```

**TIP**  Name your targets with identifiers.
M5004: Illegal Macro Reference

Message Type

[ERROR]

Description:
You used a name for a macro that is not an identifier. You must name all your macros with identifiers.

Example:
makeall:
   cc src.c $(***)

TIP  Name your macros with identifiers.

M5005: Macro Substitution Too Complex

Message Type

[ERROR]

Description:
Maker cannot resolve the macro in a table overflow.

TIP  Organize your makefile structure. Use template makefiles called from Maker with command-line macros as arguments.
**M5006: Macro Reference Not Closed**

**Message Type**

[ERROR]

**Description:**

Macro has no right brace to close the macro.

**Example:**

```
makeall:
    cc src.c $(MYMAC
```

**TIP**

Add a right brace.

---

**M5007: Unknown Macro: <macroname>**

**Message Type**

[ERROR]

**Description:**

Maker did not recognize `<macroname>` as a declared macro.

---

**M5008: Macro Definition or Command Line Too Long**

**Message Type**

[ERROR]

**Description:**

Maker cannot read a line in the makefile because it is too long.
M5009: Illegal Include Directive

Message Type

[ERROR]

Description:
The include directive has too many arguments.

Example (invalid):

```
INCLUDE macros.inc utils.inc
```

TIP

Divide the include into multiple includes:

```
INCLUDE macros.inc
INCLUDE utils.inc
```

M5010: Illegal Line

Message Type

[ERROR]

Description:
Maker encountered a syntax error in the makefile. The line starts with an invalid token sequence.

Example (invalid):

```
makeAll: Compile Link
  echo "-- all done  ## command has to start with spaces
```
M5011: Illegal Suffix for Inference Rule

Message Type
[ERROR]

Description:
The rule has incorrect syntax.

Example (correct):
```
.c.o :
$(CC) $(CFLAGS) $*.c
```

M5012: Include File Not Found: <includefile>

Message Type
[WARNING]

Description:
The filename given as an argument of the INCLUDE command does not specify an existing file.

TIP  Verify the correctness of the path settings in your default.env file; verify that the environment variable DefaultDir in the File MCUTOOLS.INI did not set the default directory.
M5013: Include File Too Long: <includefile>

Message Type
[ERROR]

Description:
Maker cannot include the specified file because it is too big.

TIP   Divide your included file into several smaller files.

M5014: Circular Macro Substitution in <macroname>

Message Type
[ERROR]

Description:
Maker detected a circular reference in the macro substitution.

M5015: Colon (:) Expected

Message Type
[ERROR]

Description:
Always mark a target declaration with a colon after the target identifier, followed by the dependencies.
M5016: Filename After INCLUDE Expected

Message Type
[ERROR]

Description:
Maker detected a token after the INCLUDE command that is not a filename which conforms to a Maker identifier.

TIP
Do not use non-alphanumeric characters in filenames, even if the operating system allows them.

M5017: Circular Include, File <includefile>

Message Type
[ERROR]

Description:
Maker does not allow circular include references in a makefile.

Example:
.mak file includes A.inc. A.inc includes B.inc. B.inc includes C.inc. C.inc includes A.inc

M5018: Entry Doesn't Start at Column 0

Message Type
[ERROR]

Description:
Entries (Identifier: dependencies,) must start at the first column of a line.
M5019: No Makefile Found

Message Type
[ERROR]

Description:
The makefile specified in the argument list does not exist.

TIP
Verify the correctness of the path settings in your default.env file; also verify that the default directory, set by the DefaultDir environment variable in the MCUTOOLS.INI file, is not set.

M5020: Fatal Error During Initialization

Message Type
[ERROR]

Description:
The Maker initialization procedure failed.

TIP
Try to restore the configuration in which Maker previously worked.

M5021: Nothing to Make: No Target Found

Message Type
[ERROR]

Description:
The Maker did not specify a target.
M5022: Don't Know How to Make <target>

Message Type
[ERROR]

Description:
The target-dependency list contains an identifier that does not exist as a file and does not reside in the target list of the makefile.

TIP
This message sometimes appears even if target or file dependencies exist. Maker dependency resolutions do not always find all targets, especially when you work with multiply layered rules. For this reason, structure the makefile another way and check the settings in your default.env file.

M5023: Circular Dependencies Between <target1> and <target2>

Message Type
[ERROR]

Description:
<target1> is in the transitive closure of circular dependencies. For example, build <target1> <target1>. <target2> is the last target handled before Maker detects the circular dependencies.

Example:

```
XX:    AA BB
AA:    FF EE
BB:    DD
DD:    XX
EE:    
FF:    
## XX is dependent (transitive closure) on AA, BB, FF, EE, DD, XX
```
**M5024: Illegal Option**

**Message Type**
- **[ERROR]**

**Description:**
The option specified in the command line has an illegal format.

**Example:**
```
Maker test.mak -DCC+\HC12\CHC12.EXE
```
instead of
```
Maker test.mak -DCC=\HC12\CHC12.EXE
```

**TIP**
With `-h`, Maker prints all available options with the expected argument list.

**M5027: Making Target <target>**

**Message Type**
- **[INFORMATION]**

**Description:**
Maker currently builds the specified target.

**TIP**
The two special targets BEFORE and AFTER execute just before and after the top target. Use them for initiations and cleanup.
M5028: Command Line Too Long: <commandline>

**Message Type**

[ERROR]

**Description:**
The command line passed to Maker is too long for Maker.

M5029: Illegal Target Name: <targetname>

**Message Type**

[ERROR]

**Description:**
You specified an invalid name as the target, which can happen with several command-line arguments. Maker takes the first argument as a make file name and all remaining arguments as target names. If some target names are invalid, this message appears. If you ignore this message with the message move options, then Maker ignores the invalid target name.

Exec Process Messages

This section explains messages that can appear when a command in a target’s build-command list fails.

M5100: Command Line Too Long for Exec

**Message Type**

[ERROR]

**Description:**
The length of a command in the target’s command list in the makefile is too long to execute.
M5101: Two File Names Expected

Message Type
[ERROR]

Description:
Some Maker commands (such as Copy or Ren) need two filenames as arguments. If two filenames do not exist, this error message occurs.

M5102: Input File Not Found

Message Type
[ERROR]

Description:
A built-in file command that needs to open a source file for reading could not find that source file.

M5103: Output File Not Opened

Message Type
[ERROR]

Description:
A built-in file command that needs to open or create a destination file for writing failed to open or create that destination file.

TIP  Check the settings in your default.env file.
M5104: Error While Copying

Message Type
[ERROR]

Description:
While copying one file, another failed in the block-copy loop. Maker opened the file but the blockwise write operation failed.

TIP Check the attributes of the destination file.

M5105: Renaming Failed

Message Type
[ERROR]

Description:
Maker failed to rename a file.
Potential causes are:
• Illegal filenames as arguments
• The source file does not exist, or another process is using it
• The destination file name is already in use.

TIP Check the file (including its attributes) to rename the file.
M5106: File Name Expected

Message Type
ERROR

Description:
Maker expects an argument of a built-in command to specify an existing file, but it has an illegal format for a file name.

TIP
Use only names and extensions allowed for C-identifiers, even if your operating system permits more character types for filenames.

M5107: File Does Not Exist

Message Type
ERROR

Description:
Maker expects an argument of a built-in command to specify an existing file, but the file does not exist.

TIP
Check the settings in your default.env file.
M5108: Called Application Detected an Error

Message Type
[ERROR]

Description:
The application that Maker called detected an error not reported in detail in its error output, or Maker did not find the error output.

TIP
Use a file named EDOUT or another file that you specify using the environment variable ERRORFILE in your default.env file. Maker prints the lines in this file starting with ERROR, FATAL, WARNING or INFORMATION if enabled in the Maker.

M5109: Echo <commandline>

Message Type
[INFORMATION]

Description:
This message appears when Maker calls an application. The entire macro-expanded command line displays.

M5110: Called Application Caused a System Error

Message Type
[ERROR]

Description:
The program that Maker executed exited with an operating-system error.
M5111: Change Directory (cd) Failed

Message Type

[ERROR]

Description:
The built-in cd command could not change the directory.

TIP
Make sure the specified directory exists. Check your working directory when using relative paths.

M5112: Called Application: <error>

Message Type

[ERROR]

Description:
The called application detected an error and wrote to it the error-output file. Maker prints the error message if you enable the message type in Make.

Example:

ERROR M5112:
called application detected an error:
ERROR C1005: “Illegal storage class!”
The string quoted is the called program’s message.

TIP
Try to run the application manually with the specified arguments. You can reclassify the called program’s message class to ERROR, WARNING, or INFORMATION; you can also disable it.
M5113: Called Application: <warning>

Message Type
[WARNING]

Description:
The called application issued a warning and wrote it to the error output file. Maker prints the warning message if you enable its message type in Make.

Example:
WARNING M5113:
called application:
"WARNING C1038: Cannot be friend of myself"
The string quoted is the called program's message. If you classify M5113 as an error, Maker prints message as:
ERROR M5112: called application: "WARNING C1038: Cannot be friend of myself"

TIP Try to run the application manually with the specified arguments. You can reclassify the called program’s message class to ERROR, WARNING, or INFORMATION; you can also disable it.

M5114: Called Application: <information>

Message Type
[INFORMATION]

Description:
The called application issued information and wrote it to its error output file. Maker prints the information message.

Example:
INFORMATION M5114:
called application:

“INFORMATION C1390: Implicit virtual function”

The string quoted is the called program’s message.

**TIP**  Try to run the application manually with the specified arguments. You can reclassify the called program’s message class to ERROR, WARNING, or INFORMATION; you can also disable it.

---

**M5115: Called Application: <fatal>**

**Message Type**

[ERROR]

**Description:**

The called application detected a fatal error and wrote the message to its error output file. Maker prints the fatal warning message if you enabled that message type in Maker.

**Example:**

```
ERROR M5115:
called application:
“FATAL C1403: Out of memory”
The string quoted is the called program’s message.
```

**TIP**  Try to run the application manually with the specified arguments. You can reclassify the called program’s message class to ERROR, WARNING, or INFORMATION; you can also disable it.
M5116: Could Not Delete File

Message Type
[WARNING]

Description:
The built-in `del` command could not delete the specified argument file.

TIP Make sure that the file to delete exists and that its attributes allow Maker to delete it.

M5117: Path Was Not Found

Message Type
[ERROR]

Description:
Maker could not restore an old directory that changed with the built-in command `cd` after the end of the command-list scope.

TIP Check whether the old directory exists. It must exist if you use `cd`.

M5118: Could Not Create Process: `<diagnostic>`

Message Type
[ERROR]

Description:
The operating system issues this message when the called process cannot run. The detailed message resides in `<diagnostic>`.
M5119: Exec <commandline>

Message Type
[INFORMATION]

Description:
Maker issues this message after it calls an application. The entire macro-expanded command line displays.

M5120: Running Version with Limited Number of Execution Calls. Number of Allowed Execution Calls Exceeded.

Message Type
[FATAL]

Description:
This message does not appear when you have a fully registered version of Maker. A nonregistered demonstration version has processing limitations. The demonstration version has a limit of 5 command calls. If you exceed this limit in one run, M5120 appears and the make process stops.

M5121: The Files <file1> and <file2> Are Not Identical

Message Type
[INFORMATION]

Description:
A fc or fctext built-in command detected that two files are not identical.
M5122: The Files <file1> and <file2> Are Identical

Message Type
[INFORMATION]

Description:
A fc or fctext built-in command detected that two files are identical.

M5153: Processing Make Files Under Win32s Is Not Supported by the Maker

Message Type
[FATAL]

Description:
Maker cannot synchronize the execution of commands with its own processing under Win32s and cannot run under Win32s. This error occurs because a 32-bit application running under Win32s with the 32-bit API cannot detect a completed called application. The Maker issues this error message if you try to run a makefile under Win32s and stops execution.
Modula-2 Maker Messages

This section explains messages that can appear when the build process for Modula-2 fails.

M5700: Environment Variable COMP Not Set

Message Type
[ERROR]

Description:
The COMP environment variable defines the Modula-2 compiler. When you do not set this variable, the Modula-2 Maker can run only in silent mode (option -s).

M5701: Environment Variable LINK Not Set

Message Type
[ERROR]

Description:
The LINK environment variable defines the linker. When you do not set this variable, the Maker can run only in silent (option -s) or in compile-only mode (option -c).

M5702: Neither Source Nor Symbol File Found: <source file>

Message Type
[ERROR]

Description:
The compiler found neither the object file nor the source file. It could not build the target.

TIP Check the settings in your default.env file.
M5703: Circular Imports in Definition Modules

**Message Type**

[ERROR]

**Description:**
The transitive closure of a module’s import list includes the module itself (a circular dependency list).

**TIP**
Layer your application and put basic types included from different layers into separate modules.

M5704: Can't Recompile <source file> (No Source Found)

**Message Type**

[ERROR]

**Description:**
The compiler could not find the specified source file.

**TIP**
Determine whether the source file exists in a location other than expected. Also check the settings in your default.env file.
M5705: No Make File Generated (Top Module Not Found)

Message Type

[WARNING]

Description:

The compiler could not write the makefile for the Modula-2 project because you did not specify the top target.

TIP  Check the settings in your default.env file.

M5706: Couldn't Open the Listing File <list file>

Message Type

[WARNING]

Description:

A file error occurred upon opening or closing the listing file for Modula-2 Make.

TIP  Check the settings in your default.env file.
M5708: Couldn't Open the Makefile

Message Type
[ERROR]

Description:
The makefile does not exist, or the make process could not open it for reading.

TIP  The default extension for Modula-2 makefiles is .MOD.

TIP  Check the settings in your default.env file.

M5761: Wrote Makefile <makefile>

Message Type
[INFORMATION]

Description:
Maker prints this information if no error occurred and the Modula-2 Maker succeeded in creating the makefile.

M5762: Wrote Listing File <listfile>

Message Type
[INFORMATION]

Description:
Maker prints this information if no error occurred and the Modula-2 Maker succeeded in creating the listing file.
M5763: Compilation Sequence

Message Type

[INFORMATION]

Description:

Announces the print listing to the Maker standard output instead of to a file listing.
Using the Linux Command Line Programs

The Linux version of the HC12 build tools are ran from a shell command line. The programs are located in the prog subfolder of the CodeWarrior installation path. The programs can be run from a shell command line or specified in a makefile. Program names are:

- burner
- decoder
- libmaker
- linker
- maker

Command Line Arguments

Enter <program name> -h (for example burner -h) to display a list of available arguments and options. Program options are documented in the chapters “Burner Options”, “Decoder Options”, “Libmaker Options”, “Maker Options”, and “SmartLinker Options”. Several options are available for all programs, such as setting the formatted output of messages and specifying text color for different types of messages.

Command Examples

The following examples demonstrate some simple uses of the linux version of the HC12 command line burner, decoder, libmaker, maker, and smartlinker. Not all options or variations of options are provided.

Burner

Configuring S-Record

To use commands placed in a batch burner language file, enter:

burner -F myfile.bbl -Ns=p
**Decoding**

**Write Disassembly Listing with Source Code**
To see the assembly code intermixed with the source code enter:
```
decoder main.o -omain.lst
```

**Decode DWARF Sections**
To decode DWARF sections and create listing file enter:
```
decoder main.o -D
```

**Libmaker**

**Specifying Libmaker Commands**
You can pack arguments into the `-Cmd` option and pass it to the libmaker command directly. For example:
```
libmaker -cmd "a.o+b.o=c.lib"
```
Also refer to “`-Mar: Freescale Archive Commands`”

**Setting Color of Error Messages**
The color setting options such as `-WmsgCE` are available for the Windows operating system only.

**Linker**

**Adding Object Files**
Enter the following command to add object files.
```
linker example.prm -Add{objfile1.o objfile2.o}
-Add{lib1.lib}
```

**Generate an S-Record File**
To link and allocate object files (in ELF format) and create a S-record file enter:
```
linker -Fe linker.prm -B
```
Maker

Defining a Macro

To define a macro in a makefile that identifies the compiler to be used enter:

```
maker -dCOMP=chc12.exe -m mymake.mak
```

Using a Makefile

The `make` command allows you to control and define the build process. The `make` program reads a file called `makefile` or `Makefile`. This file determines the relationships between the source, object and executable files.

Once you have created your `Makefile` and your corresponding source files, you are ready to use the GNU `make` command. If you have named your `Makefile` either `Makefile` or `makefile`, `make` will recognize it. If `make` does not recognize your `makefile` or it uses a different name, you can specify `make -f mymakefile`. The order in which dependencies are listed is important. If you simply type `make` and then return, `make` will attempt to create or update the first dependency listed.
Index

Symbols

% (ENV) 452, 518
%" 452, 518
%’ 452, 518
%E 452, 518
%e 452, 518
%f 452, 518
%N 452, 518
%n 452, 518
%p 452, 518
+ 412
.ABS 29, 283
.abs 29, 31, 94
.bbl 331
.c 622
.checksum section 191
.copy 242, 252
data 242, 243
.hidefaults 368, 393, 394, 398, 440, 441, 445, 446, 450, 505, 509, 510, 629, 630, 640, 641
.ini 38, 420
.init 243
.LST 413
.lst 514
.map 94, 265
.o 513, 622
.overlap 172, 244
.prm 93, 513
.rodata 242
.rodata1 242
.s1 94
.s2 94
.s3 94
.stack 242, 243
.startData 242, 243, 252
.sx 94
.text 242, 243
/wait 414
? 626
__DEFAULT_SEG_CC__ 186
__INTERSEG_CC__ 186
__INTRAPAGE__ 185

_A 658
  Print Full Listing 519
About Box 55, 439
Absolute File 29, 31, 94, 214, 218, 513
ABSPATH 80, 199, 432
>Add 100
ALIGN 231
-Alloc 101
Application
  Startup (also see Startup) 251
-AsROMLib 103
Assembly
  Application 192
  LINK_INFO 196
  Prm File 192
  Smart Linking 193
AUTOLOAD 201
Automatic Distribution 184
Automatic structure detection 191

_B 103
-B 103
B1 unknown message 401
Batch burner 297
batch file 414
baudRate 304
Building your own Libraries 649

built-in command 625
? 626
copy 625
del 625
echo 625
fc 626
fctext 626
puts 625
rehash 627
ren 627

Built-In Commands 625

BURNER 288, 383
burner 29
Burner command files 298
Burner Dialog 383
BurnerBaudRate 388
BurnerByteCommands 387
BurnerDataBits 386
BurnerDataBus 385
BurnerDestination 384
BurnerFormat 385
BurnerHeaderFile 388
BurnerInputFile 389
BurnerLength 384
BurnerOrigin 384
BurnerOutputFile 388
BurnerOutputType 386
BurnerParity 387
BurnerSwapByte 383
BurnerUndefByte 383
busWidth 304

C
-C 659
  Write Disassembly Listing With Source Code 520
  -CAlocUnusedOverlap 104
Case Sensitivity 616
cd 625
CHECKKEYS 205
CHECKSUM 202
Checksum Computation 189

-Ci 105
ClientCommand 428
CLOSE 305
-Cmd 453
-Cocc 106
CODE 157
CodeWright 427
color 127, 128, 129, 344, 345, 346, 347, 469, 470, 471, 541, 542, 543, 544, 673, 674, 675, 676

Command
AUTOLOAD 201
CHECKKEYS 205
CHECKSUM 202
DATA 205
DEPENDENCY 206
ENTRIES 164, 167, 210, 267
HAS_BANKED_DATA 211
HEXFILE 212
INIT 213, 267
LINK 119, 199, 213, 267
MAIN 215, 267
MAPFILE 116, 215
NAMES 167, 168, 199, 218
OVERLAP_GROUP 219
PLACEMENT 158, 199, 221, 243, 247
PRESTART 223
SECTIONS 155, 223
SECTIONS 149, 199, 227
STACKSIZE 234
STACKTOP 236
START 237
VECTOR 163, 237

Command Line 627
Command line option 44
Commands 617
commands 625
comment 619
Comments 617
Common Code 233
COMP 639
Compiler
  Configuration 420
  Error
    Messages 438
Error messages 399, 451, 489, 561
Graphic Interface 408
Menu 421
Menu Bar 419
Messages 436
Option 434
Option Settings Dialog 434
Status Bar 418
Tool Bar 417
User Interface 408
concatenation of macros 620
Configuration
default.env 651
WinEdit 650
Configuration file example
project.ini 77
COPY 246, 256
copy 625
COPYRIGHT 81
-CRam 106
CTRL-S 433
Current Directory 368, 393, 441, 508, 630, 640
CurrentCommandLine 71, 377

D
-D 329, 659
-Decode DWARF Sections 521
D1 563
D1000 566
D1001 566
D2 563
D50 563
D51 564
D52 564
D64 564
D65 565
D66 566
DATA 205
dataBit 306
d D E
DDE option 45
Decoder 29, 503
  Error Feedback 587
  Error messages 585
  Input File 513, 517, 561, 586
  Messages 583
  Status Bar 575
  Tool Bar 574
Default Directory 370, 632
DEFAULT.ENV 68, 368, 393, 394, 398, 440,
  441, 445, 446, 450, 505, 509, 510, 629, 630, 640,
  641
default.env 651
DEFAULT_RAM 246, 247
DEFAULT_ROM 246, 247
DEFAULTDIR 62, 82, 393, 444, 630, 632
DefaultDir 370, 632
del 625
Dependencies 617
DEPENDENCY 206
Dependency 95
destination 306
directives 624
-Disp 660
-Dist 107
-DistFile 107
-DistInfo 108
-DistOpti 109
DISTRIBUTE_INTO 187
Distribution Segment 186
-DistSeg 109
DO 307
DOS 414
DOS prompt 414
Dynamic Macros 621

E
-E 660
-Decode ELF sections 524
-E Application entry point 110
EBNF 499
ECHO 308
echo 625
-Ed
  Dump ELF sections in LST File 526
  Editor 43, 44, 375, 635
  Editor section 69
  Editor_Exe 66, 70, 373, 376, 633, 635
  Editor_Name 69, 373, 375, 632, 635
Editor_Opts 66, 70, 373, 376, 633, 635
EditorCommandLine 381
EditorDDEclientName 381
EditorDDESERVICEName 382
EditorDDETopicName 382
EditorType 380
EDOUT 638
ELSE 308
END 309
ENTRIES 164, 167, 210, 267
-Env 111, 330, 454, 506, 661
Set Environment Variable 527
ENVIRONMENT 83, 394, 440, 445
Environment
  COMP 639
  DEFAULTDIR 62, 393, 630, 632
  ENVIRONMENT 367, 394, 441
  ENVIRONMENT 440
  ERRORFILE 395, 446
  File 367, 440, 629
  FLAGS 643
  GENPATH 397, 638
  HIENVIRONMENT 394, 641
  HIPATH 397, 644
  HITEXTFAMILY 645
  HITEXTKIND 646
  HITEXTSIZE 647
  HITEXTSTYLE 648
  LINK 645
  TMP 398, 449
  Variable 367, 392, 440, 629
Environment Variable 59, 79, 444, 508, 639
  ABSSPATH 80, 199
  ABSPATH 496
  COPYRIGHT 81
  DEFAULTDIR 82, 444
  ENVIRONMENT 83, 445
  ERRORFILE 84
  GENPATH 85, 199, 448, 513
  HIENVIRONMENT 83, 445, 509
  INCLUDEDTIME 86
  LIBPATH 495
  LINKOPTIONS 99
  LINKPTIONS 87
OBJPATH 87, 199
OBJPATH 495
RESETVECTOR 88
SRECORD 89
SYMPATH 495
TEXTPATH 90, 199, 449, 514
TEXTPATH 496
TMP 91
USERNAME 92
440
Error
  Messages 438
  Error File 96
  Error Format
    Microsoft 349, 350, 474, 545, 546, 678
    Verbose 349, 474, 545
Error Listing 96, 638
Error messages 399, 451, 489, 561, 585, 611, 693
ERRORFILE 84, 395, 446
Explorer 60, 408, 441
Extended Backus-Naur Form, see EBNF
F
  -F 330
    Object File Format 527
  -F object file format 111
F2 417
fc 626
fctext 626
File
  Absolute 29, 31, 94, 214, 218, 513
  Environment 367, 440, 629
  Error 96
  Intel Hex 514
  Library 218
  Map 94, 214, 216, 265, 496
  Object 93, 218, 513
  Parameter 93
  Parameter (Linker) 197
  S 94
  SRecord 514
File Manager 60, 441
FILL 232
FLAGS 643
FOR 310
format 311

**G**
GENPATH 85, 199, 397, 448, 638
Global editor option 41
Group 370, 631
GUI Graphical User Interface 408

**H**
-H 331, 455, 662
List options 112
Prints the List of All Available Options 528
HAS_BANKED_DATA 211
header 312
HEXFILE 212
HIENVIRONMENT 83, 394, 445, 509, 641
HIPATH 397, 644
HITEXTFAMILY 645
HITEXTKIND 646
HITEXTSIZE 647
HITEXTSTYLE 648
HOST 328

**I**
-I 662
IBCC_FAR 187
IBCC_NEAR 187
Icon 408
IDF 440, 441
IF 312
Implementation Restriction 627
INCLUDETIME 86
Inference Rules 622
inference rules 622
INIT 213, 267
INPUT 328
Intel Hex 528
Intel Hex File 514

**L**
-L 113, 663
Produce inline assembly file 529
len 313
Libmaker 29, 407
LIBPATH 432
Libraries 407
Library
Adding own Objects 652
Object 412
Library File 218
-Lic 113, 332, 456, 459, 663
Print license information 530
-LicA 333, 457, 664
License Information about every
Feature 530
-Lica 114
-LicBorrow 114, 458, 664
Borrow license feature 531
-LicWait 115, 333, 335, 665
Wait for floating license from floating
license server 532
Line Breaks 616
Line Continuation 391, 443, 637
LINK 119, 199, 213, 267, 645
Linker
Configuration 38
Input File 93
Menu 40
Menu Bar 38
Message Settings Dialog Box 52
Messages 52
Options 50
Output Files 94, 514
Status Bar 38
Tool Bar 37
Linker parameter file 269
LINKOPTIONS 87
local option 43

**M**
-M 116, 666
Macro
| circular definition | 618 |
| definition | 618 |
| in Make Files | 618 |
| redefinition | 618 |
| reference | 618 |
| static macro | 618 |
| macro comments in macros | 619 |
| concatenation | 620 |
| dynamic macro | 621 |
| Macros | 618 |
| MAIN | 215, 267 |
| Makefiles |  |
| Case Sensitivity | 616 |
| Comments | 617 |
| Line Breaks | 616 |
| restrictions | 627 |
| Structured Makefiles | 654 |
| Syntax | 616 |
| makefiles include | 624 |
| Maker |  |
| Advanced Options Dialog | 607 |
| Building your own Libraries | 649 |
| Error Feedback | 614 |
| Error messages | 611, 693 |
| Input File | 613, 638 |
| Messages | 609 |
| Status Bar | 599 |
| Tool Bar | 598 |
| maker | 589 |
| Maker utility | 29 |
| Making C Applications | 615 |
| map | 214 |
| Map File | 94, 214, 216, 265, 496 |
| COPYDOWN | 266 |
| DEPENDENCY TREE | 266 |
| FILE | 265 |
| OBJECT ALLOCATION | 265 |
| OBJECT DEPENDENCY | 266 |
| SEGMENT ALLOCATION | 265 |
| STARTUP | 265 |
| STATISTICS | 266 |
| TARGET | 265 |
| UNUSED OBJECTS | 266 |
| Map file | 270 |
| MAPFILE | 116, 215 |
| -Mar | 460 |
| MCUTOOLS.INI | 41, 61, 369, 424, 577, 601, 631 |
| MESSAGE | 328 |
| Message list | 401 |
| MESSAGES | 328 |
| Messages Settings | 436, 583, 609 |
| Microsoft | 349, 350, 474, 545, 546, 678 |
| Microsoft Developer Studio | 428 |
| - MkAll | 666 |
| msdev | 428 |
| N |  |
| -N | 117, 336, 460, 667 |
| Display Notify Box | 532 |
| NAMES | 167, 168, 199, 218 |
| NO_INIT | 151, 157, 224, 228 |
| - NoBeep | 117, 337, 461, 667 |
| No Beep in Case of an Error | 533 |
| - NoCapture | 668 |
| - NoEnv | 118, 337 |
| Do not use Environment | 534 |
| - NoPath | 462 |
| - NoSym |  |
| No Symbols in Disassembled Listing | 534 |
| -Ns | 338 |
| O |  |
| -O | 118, 669 |
| Defines Listing File Name | 535 |
| Object |  |
| Library | 412 |
| Object File | 93, 218, 513 |
| OBJPATH | 87, 199, 432 |
| - OCopy | 119 |
| OPENCOM | 314 |
| OPENFILE | 315 |
| Option |  |
| HOST | 328 |
| INPUT | 328 |
| MESSAGE | 328 |
| MESSAGES | 328 |
VARIOUS 328
Option Settings Dialog 50
Options 370, 380, 632, 657
Groups 657
Options section 62
origin 316
OVERLAP_GROUP 219
Overlapping Locals 170
OVERLAYS 169

P
PAGED 151, 157, 169, 224, 228
Parameter
File (Linker) 197
Parameter File 93
parity 316
Partial fields 192
Path 369, 631
path in S0 record 338
Path List 77, 390, 442, 506, 637
PAUSE 324
piper 414
piper.exe 414
PLACE 158, 199, 221, 243, 247
Premia 427
PRESTART 223
Pma file controlled Checksum Computation 190
-Proc
  Set Processor 536
Processing 617
-Prod 69, 120, 339, 375, 412, 463
Program Startup (also see Startup) 251
Project Directory 60
project.ini 69, 288, 375
puts 625

Q
Qualifier 151, 157, 227
  CODE 157
  NO_INIT 151, 157, 224, 228
  PAGED 151, 157, 224, 228
  READ_ONLY 151, 157, 224, 227
  READ_WRITE 151, 157, 224, 227

R
READ_ONLY 151, 157, 224, 227
READ_WRITE 151, 157, 224, 227
REALLOC_OBJ 186
RecentCommandLine 377
rehash 627
Release Notes 504
RELOCATE_TO 230
ren 627
Restriction
  Implementation 627
restrictions 627
RGB 127, 128, 129, 130, 344, 345, 346, 347, 469,
  470, 471, 541, 542, 543, 544, 637, 673, 674, 675, 676
ROM Library 267
ROM library 213, 218, 252, 256
ROM_LIB 214, 267
ROM_VAR 246
Rules 622
rules 622
Runtime support 192

S
  -S 121, 670
  S File 94
  S0 338
  S1 338
  S2 338
  S3 338
  S7 338
  S8 338
  S9 338
  SaveAppearance 371
  SaveEditor 371
  SaveOnExit 371
  SaveOptions 372
  Section 241, 245
    .copy 242, 252
    .data 242, 243
    .init 243
    .overlap 244
    .rodata 242
    .rodata1 242
.stack 242, 243
.startData 242, 243, 252
.text 242, 243
Pre-defined 242
Qualifier 157
rodata 242
SECTIONS 155, 223
Sections
using 249
Segment 241
._OVERLAP 248
._PRESTART 247
Alignment 153, 227, 231
COPY 246, 256
DEFAULT_RAM 246
DEFAULT_RAM 247
DEFAULT_ROM 246, 247
Fill Pattern 154, 232
dfill pattern 227
Optimizing Constants 233
Pre-defined 246
Qualifier 151, 227
Relocation 227, 230
ROM_VAR 246
ST STACK 246, 247
STARTUP 246, 247, 256
STRINGS 246
SECTIONS 149, 199, 227
SENDBYTE 317
SENDWORD 318
Service Name 428
-SFixups 121
-ShowAboutDialog 411
-ShowBurnerDialog 411
ShowConfigurationDialog 411
-ShowMessageDialog 411
-ShowOptionDialog 411
-ShowSmartSliderDialog 411
ShowTipOfDay 380
SLINELEN 319
SmartLinking 32, 163, 165
SmartLinker 29, 31
Special Modifiers 452
SRECORD 320
S-Record 528
SRecord File 514
SSTACK 246, 247
ST. SectionTitle 283
STACK 234
STACKSIZE 234
STACKTOP 236
START 237
start 414
STARTUP 246, 247, 256
Startup
Application 251
startup 69, 375
Startup descriptor 269
Startup Function 256, 258
User Defined 256, 258
startup option 411
Startup Structure 251, 256
finiBodies 254
flags 252, 257
initBodies 254
libInits 254, 258
main 253, 257
mlnts 258
nofFiniBodies 254
nofInitBodies 254
nofLibnits 254
nofZeroOuts 253, 257
pZeroOut 253, 257
stackOffset 253, 257
toCopyDownBeg 253, 257
User Defined 255
-StatF 122
StatusBarEnabled 377
stderr 414
stdout 363, 414, 488
STRINGS 246
Structured Makefiles 654
swapByte 321
synchronization 414
Syntax of Makefiles 616
T
-T
Shows the Cycle Count for each
Instruction 537
Target
Dependencies 617
TEXTPATH 90, 199, 432, 449, 514
THEN 322
Tip of the Day 606
TipFilePos 379
TipTimestamp 65
TMP 91, 398, 449
TO 323
ToolBarEnabled 378
Topic Name 428

U
UltraEdit 428
undefByte 324
UNIX 441, 630
UNIX Make 616
USERNAME 92
Using Makefiles 616

V
-V 122, 340, 341, 463, 670, 671
    Prints the Decoder Version 538
Variable
    Environment 367, 440, 629
VARIOUS 328
VECTOR 163, 237
Vector 32
Vector table
    initialize 275
-View 123, 340, 464
    Application Standard Occurrence (PC) 538

W
-W1 124, 364, 465
    No Information Messages 557
-W2 124, 364, 466
    No Information and Warning Messages 558
-WErrFile 125, 343, 468
    Create "err.log" Error File 539
WindowFont 379
WindowPos 73, 378
Windows 441, 630
WinEdit 396, 447, 650
Winedit 427
-Wmsg8x3 126, 342, 467
    Cut file names in Microsoft format to
8.3 540
-WmsgCE 127, 344, 469
    RGB color for error messages 541, 673
-WmsgCF 127, 345, 469
    RGB color for fatal messages 541, 674
-WmsgCI 128, 345, 470
    RGB color for information messages 542,
674
-WmsgCU 129, 346, 471
    RGB color for user messages 543, 675
-WmsgCW 129, 347, 471
    RGB color for warning messages 544, 676
-WmsgFb 130, 348, 472
    Set message file format for batch mode 544
-WmsgFbi 348, 472
-WmsgFbm 348, 472
-WmsgFi 132, 349, 474
    Set message format for interactive mode 545
-WmsgFim 349, 474
-WmsgFiv 349, 474
-WmsgFob 133, 351, 475, 546
    Message format for Batch Mode 546
-WmsgFoi 352, 477
    Message Format for Interactive Mode 548
-WmsgFon 137, 353, 478
    Message Format for no File Information 549
-WmsgFonf 138, 355, 479
    Message Format for no Position
Information 550
-WmsgNe 140, 356, 480
    Number of Error Messages 551
-WmsgNi 140, 357, 481
    Number of Information Messages 551
-WmsgNu 141, 357, 482
    Disable User Messages 552
-WmsgNw 142, 358, 483
    Number of Warning Messages 553
-WmsgSd 143, 359, 484

HC(S)08/RS08 and HC(S)12 Build Tools Utilities Manual 737
Setting a Message to Disable 553
-WmsgSe 144, 360, 484
  Setting a Message to Error 554
-WmsgSi 145, 360, 485
  Setting a Message to Information 555
-WmsgSw 145, 361, 486
  Setting a Message to Warning 555
-WOutFile 146, 362, 487
  Create Error Listing File 556
-WStdout 147, 363, 488
  Write to standard output 556

X
-X
  Write disassembled listing only 558

Y
-Y
  Write disassembled listing with source and all comments 559