MCUez

Easy development software from the company that knows MCU hardware best

MCUez HC05 Assembler User's Manual
MCUEZASM05/D
Rev. 1

MOTOROLA

MCUEZASM05/D
Rev. 1

MCUez HC05 Assembler

User's Manual

MOTOROLA

Important Notice to Users

While every effort has been made to ensure the accuracy of all information in this document, Motorola assumes no liability to any party for any loss or damage caused by errors or omissions or by statements of any kind in this document, its updates, supplements, or special editions, whether such errors are omissions or statements resulting from negligence, accident, or any other cause. Motorola further assumes no liability arising out of the application or use of any information, product, or system described herein; nor any liability for incidental or consequential damages arising from the use of this document. Motorola disclaims all warranties regarding the information contained herein, whether expressed, implied, or statutory, *including implied warranties of merchantability or fitness for a particular purpose*. Motorola makes no representation that the interconnection of products in the manner described herein will not infringe on existing or future patent rights, nor do the descriptions contained herein imply the granting or license to make, use or sell equipment constructed in accordance with this description.

The computer program contains material copyrighted by Motorola, Inc., first published in 1997, and may be used only under a license such as the License For Computer Programs (Article 14) contained in Motorola's Terms and Conditions of Sale, Rev. 1/79.

Trademarks

This document includes these trademarks:

MCUez and MCUasm are trademarks of Motorola, Inc.

Microsoft Windows, Word, Win32, and Microsoft Developer Studio are registered trademarks of Microsoft Corporation in the U.S. and other countries.

UNIX is a registered trademark of the Open Group in the U.S. and other countries.

WinEdit is a trademark of Wilson WindowWare.

P&E is a trademark of P&E Microcomputer Systems, Inc.

List of Sections

Section 1. General Information 23

Section 2. Graphical User Interface. 37

Section 3. Environment Variables 55

Section 4. Files 71

Section 5. Assembler Options 77

Section 6. Sections 107

Section 7. Assembler Syntax 117

Section 8. Assembler Directives 149

Section 9. Macros 195

Section 10. Assembler Listing File 201

Section 11. Operating Procedures 209

Section 12. Assembler Messages 235

Appendix A. MASM Compatibility 295

Appendix B. MCUasm Compatibility 299

Appendix C. P&E Converter. 301

Index. 325

Table of Contents

Section 1. General Information

1.1 Contents 23

1.2 Introduction. 23

1.3 Structure of This Manual 24

1.4 Getting Started 24

1.4.1 Creating a New Project 25

1.4.2 Creating an Assembly Source File. 27

1.4.3 Assembling a Source File 28

1.4.4 Linking an Application 33

Section 2. Graphical User Interface

2.1 Contents 37

2.2 Introduction. 37

2.3 Starting the Motorola Assembler 38

2.4 Assembler Graphical Interface 39

2.4.1 Window Title 39

2.4.2 Content Area 40

2.4.3 Assembler Toolbar. 41

2.4.4 Status Bar 42

2.4.5 Assembler Menu Bar 42

2.4.6 File Menu 43

2.4.6.1 Editor Settings Dialog 44

2.4.6.2 Save Configuration Dialog 49

2.4.7 Assembler Menu 50

2.4.8 View Menu. 50

2.4.8.1 Option Settings Dialog Box 51

2.4.9 Specifying the Input File 52

2.4.9.1 Using the Editable Combo Box in the Toolbar. 52

2.4.9.2 Using the Entry File | Assembly 53

2.4.9.3 Using Drag and Drop 53

2.5 Error Feedback 53

Section 3. Environment Variables

3.1 Contents . . . 55

3.2 Introduction . . . 56

3.3 Paths . . . 57

3.4 Line Continuation . . . 57

3.5 Environment Variables Description . . . 58
- 3.5.1 ASMOPTIONS . . . 59
- 3.5.2 GENPATH . . . 60
- 3.5.3 ABSPATH . . . 61
- 3.5.4 OBJPATH . . . 62
- 3.5.5 TEXTPATH . . . 63
- 3.5.6 SRECORD . . . 64
- 3.5.7 ERRORFILE . . . 65
- 3.5.8 COPYRIGHT: Copyright Entry in Object File . . . 67
- 3.5.9 INCLUDETIME: Creation Time in Object File . . . 68
- 3.5.10 USERNAME: User Name in Object File . . . 69

Section 4. Files

4.1 Contents . . . 71

4.2 Introduction . . . 71

4.3 Input Files . . . 71
- 4.3.1 Source Files . . . 72
- 4.3.2 Include Files . . . 72

4.4 Output Files . . . 72
- 4.4.1 Object Files . . . 72
- 4.4.2 Absolute Files . . . 73
- 4.4.3 Motorola S Files . . . 73
- 4.4.4 Listing Files . . . 73
- 4.4.5 Debug Listing Files . . . 74
- 4.4.6 Error Listing Files . . . 74

Section 5. Assembler Options

5.1 Contents . . . 77

5.2 Introduction . . . 78

5.3 ASMOPTIONS . . . 78

5.4 Assembler Options 79
5.4.1 -CI 81
5.4.2 -Env 82
5.4.3 -F2 -FA2 83
5.4.4 -H 84
5.4.5 -L 85
5.4.6 -Lc 87
5.4.7 -Ld 89
5.4.8 -Le 91
5.4.9 -Li 93
5.4.10 -Ms -Ml 95
5.4.11 -MCUasm 96
5.4.12 -N 97
5.4.13 -V 98
5.4.14 -W1 99
5.4.15 -W2 100
5.4.16 -WmsgNe 101
5.4.17 -WmsgNi 102
5.4.18 -WmsgNw 103
5.4.19 -WmsgFbv -WmsgFbm 104
5.4.20 -WmsgFiv -WmsgFim 106

Section 6. Sections

6.1 Contents 107
6.2 Introduction 107
6.3 Section Attributes 108
6.3.1 Data Sections 108
6.3.2 Constant Data Sections 108
6.3.3 Code Sections 108
6.4 Section Types 109
6.4.1 Absolute Sections 109
6.4.2 Relocatable Sections 111
6.4.3 Relocatable versus Absolute Section 114
6.4.3.1 Modularity 114
6.4.3.2 Multiple Developers 114
6.4.3.3 Early Development 115
6.4.3.4 Enhanced Portability 115
6.4.3.5 Tracking Overlaps 115
6.4.3.6 Reusability 115

Section 7. Assembler Syntax

7.1 Contents 117

7.2 Introduction. 118

7.3 Comment Line 118

7.4 Source Line 119
7.4.1 Label Field 119
7.4.2 Operation Field 120
7.4.2.1 Instructions 120
7.4.2.2 Directives 125
7.4.2.3 Macro Names 125
7.4.3 Operand Fields 125
7.4.3.1 Inherent 126
7.4.3.2 Immediate 126
7.4.3.3 Direct. 127
7.4.3.4 Extended 128
7.4.3.5 Indexed, No Offset 129
7.4.3.6 Indexed, 8-Bit Offset. 130
7.4.3.7 Indexed, 16-Bit Offset. 131
7.4.3.8 Relative 132
7.4.4 Comment Field. 132

7.5 Symbols 132
7.5.1 User-Defined Symbols. 133
7.5.2 External Symbols 134
7.5.3 Undefined Symbols 134
7.5.4 Reserved Symbols 135

7.6 Constants. 135
7.6.1 Integer Constants 135
7.6.2 String Constants 136
7.6.3 Floating-Point Constants 136

7.7 Operators. 136
7.7.1 Addition and Subtraction Operators (Binary) 136
7.7.2 Multiplication, Division, and Modulo Operators (Binary) 137
7.7.3 Sign Operators (Unary) 137
7.7.4 Shift Operators (Binary). 138
7.7.5 Bitwise Operators (Binary) 138
7.7.6 Bitwise Operators (Unary). 139
7.7.7 Logical Operators (Unary). 139
7.7.8 Relational Operators (Binary) 140
7.7.9 HIGH Operator 141

7.7.10 LOW Operator 141
7.7.11 Force Operator (Unary) 142
7.8 Expressions 144
7.8.1 Absolute Expressions 145
7.8.2 Simple Relocatable Expression 146
7.9 Translation Limits 147

Section 8. Assembler Directives

8.1 Contents 149
8.2 Introduction 151
8.3 Directive Overview 151
8.3.1 Section Definition Directives 151
8.3.2 Constant Definition Directives 151
8.3.3 Data Allocation Directives 152
8.3.4 Symbol Linkage Directives 152
8.3.5 Assembly Control Directives 152
8.3.6 Listing File Control Directives 153
8.3.7 Macro Control Directives 154
8.3.8 Conditional Assembly Directives 154
8.4 ABSENTRY — Application Entry Point 155
8.5 ALIGN — Align Location Counter 156
8.6 BASE — Set Number Base 157
8.7 CLIST — List Conditional Assembly 158
8.8 DC — Define Constant 160
8.9 DCB — Define Constant Block 162
8.10 DS — Define Space 163
8.11 ELSE — Conditional Assembly 164
8.12 END — End Assembly 165
8.13 ENDIF — End Conditional Assembly 166
8.14 ENDM — End Macro Definition 166
8.15 EQU — Equate Symbol Value 167
8.16 EVEN — Force Word Alignment 168
8.17 FAIL — Generate Error Message 169
8.18 IF — Conditional Assembly 172
8.19 IFCC — Conditional Assembly 173

8.20 INCLUDE — Include Text from Another File. 175
8.21 LIST — Enable Listing. 175
8.22 LLEN — Set Line Length. 177
8.23 LONGEVEN — Forcing Longword Alignment. 178
8.24 MACRO — Begin Macro Definition 179
8.25 MEXIT — Terminate Macro Expansion 180
8.26 MLIST — List Macro Expansions 182
8.27 NOLIST — Disable Listing 185
8.28 NOPAGE — Disable Paging 186
8.29 ORG — Set Location Counter 186
8.30 OFFSET — Create Absolute Symbols 186
8.31 PAGE — Insert Page Break 188
8.32 PLEN — Set Page Length 189
8.33 SECTION — Declare Relocatable Section 189
8.34 SET — Set Symbol Value 191
8.35 SPC — Insert Blank Lines 192
8.36 TABS — Set Tab Length 192
8.37 TITLE — Provide Listing Title 192
8.38 XDEF — External Symbol Definition 193
8.39 XREF — External Symbol Reference 194

Section 9. Macros

9.1 Contents 195
9.2 Introduction. 195
9.3 Macro Overview 195
9.4 Defining a Macro 196
9.5 Calling Macros 197
9.6 Macro Parameters 197
9.7 Labels Inside Macros 198
9.8 Macro Expansion 199
9.9 Nested Macros 200

Section 10. Assembler Listing File

10.1 Contents 201
10.2 Introduction. 201
10.3 Page Header 202
10.4 Source Listing 202
10.4.1 Absolute (Abs.) Listing 203
10.4.2 Relative (Rel.) Listing 204
10.4.3 Location (Loc.) Listing 205
10.4.4 Object (Obj.) Code. 206
10.4.5 Source Line 207

Section 11. Operating Procedures

11.1 Contents 209
11.2 Introduction. 210
11.3 Working with Absolute Sections 210
11.3.1 Defining Absolute Sections in the Assembly Source File 210
11.3.2 Linking an Application Containing Absolute Sections 212
11.4 Working with Relocatable Sections 213
11.4.1 Defining Relocatable Sections in the Assembly Source File. . . . 213
11.4.2 Linking an Application Containing Relocatable Sections 214
11.5 Initializing the Vector Table 216
11.5.1 Initializing the Vector Table in the Linker PRM File 217
11.5.2 Initializing Vector Table in Assembly Source Files
Using a Relocatable Section 219
11.5.3 Initializing the Vector Table in the Assembly Source File
Using an Absolute Section 222
11.6 Splitting an Application into Different Modules 225
11.7 Using Direct Addressing Mode to Access Symbols. 227
11.7.1 Using Direct Addressing Mode to Access
External Symbols 227
11.7.2 Using Direct Addressing Mode to Access
Exported Symbols. 227
11.7.3 Defining Symbols in the Direct Page. 228
11.7.4 Using a Force Operator 228
11.7.5 Using SHORT Sections 229
11.8 Generating an .abs File Directly 229
11.8.1 Assembler Source File 230
11.8.2 Assembling and Generating the Application 231

Section 12. Assembler Messages

12.1 Contents 235

12.2 Introduction. 237
12.2.1 Warning 237
12.2.2 Error 237
12.2.3 Fatal 237

12.3 Message Codes 237
12.3.1 A1000: Conditional Directive not Closed 238
12.3.2 A1001: Conditional Else not Allowed Here 239
12.3.3 A1051: Zero Division in Expression 240
12.3.4 A1052: Right Parenthesis Expected. 241
12.3.5 A1053: Left Parenthesis Expected. 242
12.3.6 A1054 References on Non-Absolute Objects not
Allowed When Options -FA1 or -FA2 Are Enabled 243
12.3.7 A1101: Illegal Label: Label is Reserved 244
12.3.8 A1103: Illegal Redefinition of Label 245
12.3.9 A1104: Undeclared User-Defined Symbol <symbolName> 246
12.3.10 A2301: Label is Missing 247
12.3.11 A2302: Macro Name is Missing 247
12.3.12 A2303: ENDM is Illegal 248
12.3.13 A2304: Macro Definition Within Definition 249
12.3.14 A2305: Illegal Redefinition of Instruction or Directive Name . . 250
12.3.15 A2306: Macro not Closed at End of Source 251
12.3.16 A2307: Macro Redefinition 252
12.3.17 A2308: Filename Expected 253
12.3.18 A2309: File not Found 253
12.3.19 A2310: Illegal Size Character 254
12.3.20 A2311: Symbol Name Expected 255
12.3.21 A2312: String Expected. 256
12.3.22 A2313: Nesting of Include Files Exceeds 50. 257
12.3.23 A2314: Expression Must be Absolute 257
12.3.24 A2316: Section Name Required 258
12.3.25 A2317: Illegal Redefinition of Section Name 259
12.3.26 A2318: Section not Declared. 260
12.3.27 A2320: Value too Small. 261
12.3.28 A2321: Value too Big 262
12.3.29 A2323: Label is Ignored 263
12.3.30 A2324: Illegal Base (2, 8, 10, and 16) 264
12.3.31 A2325: Comma or Line End Expected 265
12.3.32 A2326: Label is Redefined 266
12.3.33 A2327: ON or OFF Expected 267
12.3.34 A2328: Value is Truncated 267

12.3.35 A2329: FAIL Found. 268
12.3.36 A2330: String is not Allowed 269
12.3.37 A2332: FAIL Found. 270
12.3.38 A2333: Forward Reference not Allowed 271
12.3.39 A2334: Only Labels Defined in the Current Assembly Unit Can Be Referenced in an EQU Expression 272
12.3.40 A2335: Exported Absolute EQU Label is not Supported 273
12.3.41 A2336: Value too Big 274
12.3.42 A2338: <Message String> 275
12.3.43 A2341: Relocatable Section not Allowed: Absolute File is Generated 276
12.3.44 A13001: Illegal Addressing Mode. 277
12.3.45 A13005: Comma Expected 278
12.3.46 A13007: Relative Branch with Illegal Target 279
12.3.47 A13008: Illegal Expression 281
12.3.48 A13101: Illegal Operand Format 281
12.3.49 A13102: Operand not Allowed 282
12.3.50 A13106: Illegal Size Specification for HC05 Instruction 282
12.3.51 A13108: Illegal Character at the End of Line 283
12.3.52 A13109: Positive Value Expected 284
12.3.53 A13110: Mask Expected 285
12.3.54 A13111: Value Out of Range. 286
12.3.55 A13201: Lexical Error in First or Second Field 287
12.3.56 A13202: Not an HC05 Instruction or Directive. 288
12.3.57 A13401: Value Out of Range — 128...127 289
12.3.58 A13403: Complex Relocatable Expression not Supported 290
12.3.59 A13405: Code Size Per Section is Limited to 32 Kbytes 291
12.3.60 A13406: HIGH with Initialized RAM not Supported 292
12.3.61 A13407: LOW with Initialized RAM not Supported. 292
12.3.62 A13601: Error in Expression 292
12.3.63 A13602: Error at End of Expression 293

Appendix A. MASM Compatibility

A.1 Contents 295
A.2 Introduction. 295
A.3 Comment Line 295
A.4 Constants. 296
A.5 Operators. 297
A.6 Directives 298

Appendix B. MCUasm Compatibility

B.1 Contents 299
B.2 Introduction 299
B.3 Labels 299
B.4 SET Directive 300
B.5 Obsolete Directives 300

Appendix C. P&E Converter

C.1 Contents 301
C.2 Overview 302
C.3 Input Files 303
C.4 Output Files 303
C.5 Error Listing File 303
C.6 Converter Environment 303
C.7 Converting a P&E Assembler Source File 304
C.7.1 Window Title 305
C.7.2 Content Area 305
C.7.3 Toolbar 306
C.7.4 Status Bar 307
C.8 Converter Menus 307
C.8.1 File Menu 308
C.8.2 Converter Menu 309
C.8.3 View Menu 310
C.9 Defining Environment Variables 310
C.9.1 Syntax for Variable Definition 311
C.9.2 Specifying Paths in Variable Definition 311
C.9.3 Line Continuation in Variable Definition 311
C.10 Environment Variables 312
C.10.1 ENVIRONMENT 312
C.10.2 ERRORFILE 313
C.10.3 GENPATH 314
C.11 Command Line Options 314
C.11.1 -H 315
C.11.2 -V 316

C.11.3 -O 316
C.11.4 -N 317
C.12 Ambiguities in P&E Assembler Syntax 317
C.12.1 Dashes in Labels 317
C.12.2 Percent Characters in Macros 319
C.12.3 Labels in Macros 320
C.13 Modifications During Conversion 321
C.13.1 Constants 321
C.13.2 Q Suffix 321
C.13.3 T Suffix 321
C.14 Default Base 321
C.15 Directives in the First Column 321
C.16 Expressions 322
C.17 Macro Declaration 322
C.18 Operators 323
C.19 Unsupported Commands 323
C.20 Supported Commands 324

Index

Index 325

List of Figures

Figure	Title	Page
1-1	MCUez Shell	25
1-2	Environment Configuration Dialog Box	25
1-3	Working Project Directory Dialog Box	26
1-4	New Configuration Dialog Box	26
1-5	Assembler Window	28
1-6	Options Settings Dialog Box	29
1-7	Selecting an Object File Format	30
1-8	Assembling a File	31
1-9	Linker Window	34
1-10	Link Process	35
2-1	Tip of the Day Window	38
2-2	Assembler Window	39
2-3	Assembler Toolbar	41
2-4	Assembler Status Bar	42
2-5	Starting the Global Editor	44
2-6	Starting the Local Editor	45
2-7	Starting the Editor with the Command Line	46
2-8	Starting the Editor with DDE	47
2-9	Save Configuration Dialog Box	49
2-10	Option Settings Dialog Box	51
4-1	Assembler Structural Diagram	75
6-1	Absolute Section Programming Example	109
6-2	PRM File Example Code	110
6-3	Relocatable Section Programming Example	111
6-4	Defining One RAM and One ROM Area	112
6-5	Defining Multiple RAM and ROM Areas	113

List of Figures

Figure	Title	Page
7-1	Relocatable Symbols Program Example	133
7-2	SET or EQU Directive Program Example	133
7-3	External Symbol Program Example	134
7-4	Undefined Symbol Example	134
11-1	Starting the MCUez Assembler	231
11-2	Options Setting Dialog Box	232
11-3	Selecting the Object File Format	233
11-4	Generating an Assembler .abs File	234
C-1	XLATE Icon	304
C-2	Converter Window	304
C-3	Toolbar	306
C-4	Status Bar	307
C-5	Advanced Options Dialog Box	309
C-6	Command Line Options	315
C-7	Version Number and Project Directory	316
C-8	Output File Definition	316

List of Tables

Table	Title	Page
2-1	Menu Bar	42
2-2	Assembler Menu	50
2-3	Advanced Options	52
3-1	Environment Variables	58
5-1	Assembler Option Groups	79
5-2	Scope of Assembler Option Groups	79
5-3	Assembler Option Details	80
7-1	ExecuInstructions	120
7-2	Addressing Mode Notations	125
7-3	Operator Precedence	143
7-4	Expression — Operator Relationship (Unary)	146
7-5	Expression — Operator Relationship (Binary Operation)	147
8-1	Section Directives	151
8-2	Constant Directives	151
8-3	Data Allocation Directives	152
8-4	Symbol Linkage Directives	152
8-5	Assembly Control Directives	152
8-6	Assembler List File Directives	153
8-7	Macro Directives	154
8-8	Conditional Assembly Directives	154
8-9	Conditional Types	174
A-1	Operators	297
A-2	Directives	298

List of Tables

Table	Title	Page
B-1	Obsolete Directives	300
C-1	File Menu Entries	308
C-2	Supported Commands	324

Section 1. General Information

1.1 Contents

1.2 Introduction. 23

1.3 Structure of This Manual 24

1.4 Getting Started 24
1.4.1 Creating a New Project 25
1.4.2 Creating the Assembly Source File 27
1.4.3 Assembling a Source File 28
1.4.4 Linking an Application 33

1.2 Introduction

Features of the MCUez assembler include:

- Graphical user interface (GUI)
- Online help
- Support for absolute and relocatable assembler code
- 32-bit application
- Compatible with MCUasm™ release 5.3
- Conforms to Motorola assembly language input standard and *ELF/DWARF 2.0* object code format
- *ELF/DWARF 2.0* object code format

1.3 Structure of This Manual

These topics are contained in this manual:

- **Graphical user interface** — Description of the assembler GUI
- **Environment** — Description of the assembler environment variables
- **Assembler options** — Detailed description of the full set of assembler options
- **Assembler syntax** — Description of the assembler input file syntax
- **Assembler directives** — List of all directives supported by the assembler
- **Assembler messages** — Description and examples produced by the assembler
- **Appendices**
- **Index**

1.4 Getting Started

This section provides an example of how to get started with MCUez. Instructions are provided to:

- Create a new project
- Write the assembly source file
- Assemble the source file
- Link the application to generate an executable file

NOTE: *All directory paths and listings are examples. The user's paths and directory listings may change depending upon the MCUez configuration.*

1.4.1 Creating a New Project

The first step in writing an application is to define the new project. This is done in the **MCUez Shell**. Follow this sequence:

1. Start the **MCUez Shell**.

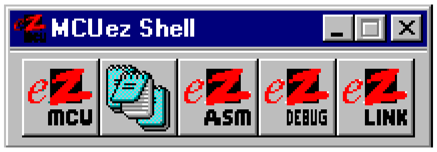
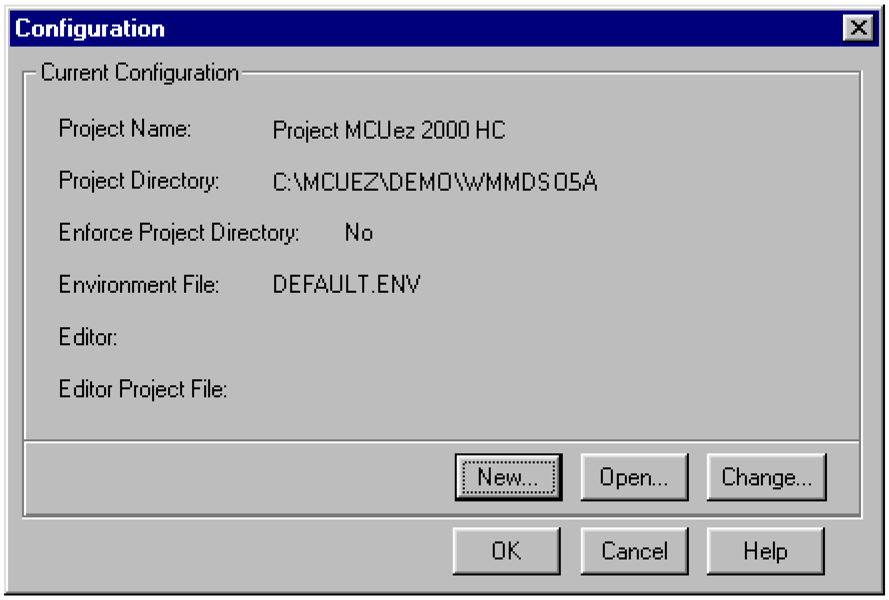


Figure 1-1. MCUez Shell

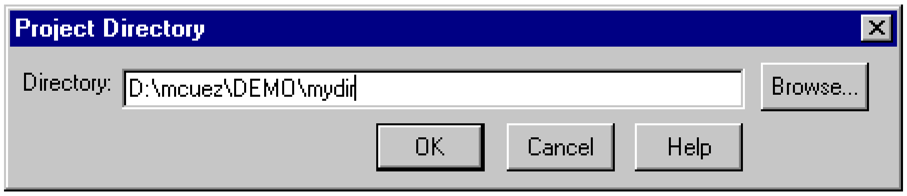
2. Click on the **ezMCU** button. The **Configuration** dialog box opens.



Project Name: Project MCUez 2000 HC
Project Directory: C:\MCUEZ\DEMO\WMMDS05A
Enforce Project Directory: No
Environment File: DEFAULT.ENV
Editor:
Editor Project File:

Figure 1-2. Environment Configuration Dialog Box

3. Click on the **New** button. The **Project Directory** dialog box opens.
4. Enter the path for the new project in the edit box. For example, substitute `C:\MCUEZ\DEMO\WMMDS05A` for `C:\MCUEZ\DEMO\mydir` as shown in **Figure 1-3**.

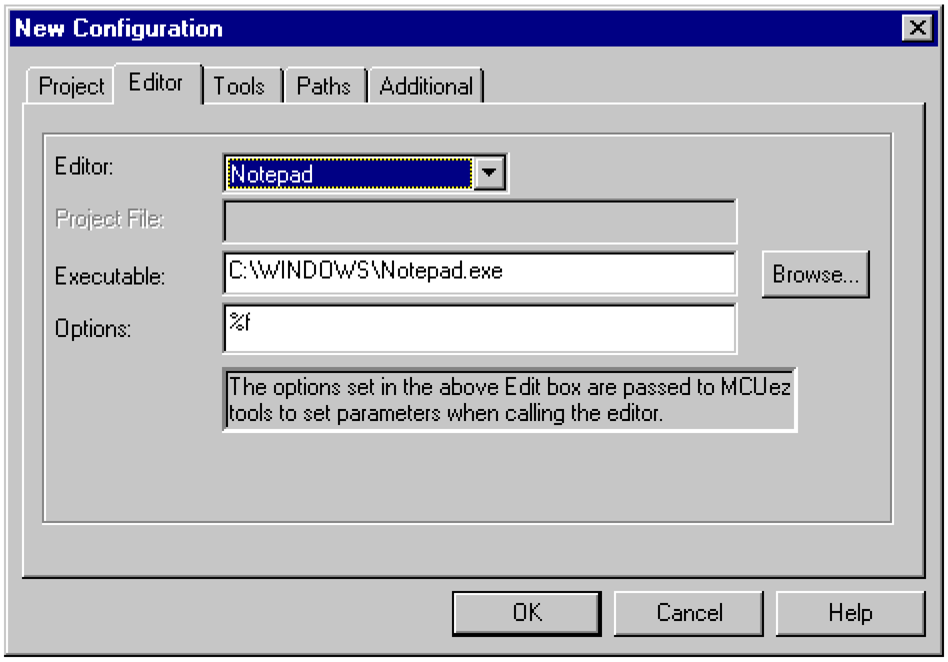


Directory: D:\mcuez\DEMO\mydir

Figure 1-3. Working Project Directory Dialog Box

NOTE: *The specified directory must be accessible from the user's PC.*

5. Click on the **OK** button.
6. The **New Configuration** dialog box is opened. Define the editor to use for the project.
7. Select the **Editor** tab. Select an editor from the **Editor** drop down box. In the **Executable** command line, enter the command used to start the editor. For example, `C:\WINDOWS\NOTEPAD.EXE`. Also, click the **Browse** button to find and select the command.



Editor: Notepad
Project File:
Executable: C:\WINDOWS\Notepad.exe
Options: %f
The options set in the above Edit box are passed to MCUez tools to set parameters when calling the editor.

Figure 1-4. New Configuration Dialog Box

8. Click on the **OK** button to automatically create the configuration files in the specified project directory.

1.4.2 Creating an Assembly Source File

After defining a new project, an application can be written. For example, the source code may be stored in a file named *test.asm* and look like this:

```
        XDEF entry          ; Make the symbol entry visible
                            ; for external module.
                            ; This is necessary to allow linker to find the
                            ; symbol and use it as the entry point for the
                            ; application.
cstSec:SECTION              ; Define a constant relocatable section
var1:DC.B 5                 ; Assign 5 to the symbol var1
dataSec:SECTION             ; Define a data relocatable section
data:DS.B 1                 ; Define one byte variable in RAM
codeSec:SECTION             ; Define a code relocatable section
entry:
RSP                         ; Reset stack pointer

LDA   var1
main:
INCA
STA   data
BRA   main
```

When writing assembly source code, pay special attention to these points:

- All symbols referenced outside the current source file (in another source file or in the linker configuration file) must be visible externally. For this reason, the assembly directive `XDEF entry` is inserted.
- To make debugging from the application easier, defining separate sections for code, constant data (defined with `DC`), and variables (defined with `DS`) is recommended. This enables the symbols located in the variable or constant data sections to be displayed in the data window component of the debugger.
- The stack pointer must be initialized when using BSR (branch to subroutine) or JSR (jump to subroutine) instructions.

1.4.3 Assembling a Source File

This step-by-step procedure describes how to assemble a source file.

1. Click on the **ezASM** button in the **MCUez Shell** to start the macro assembler. Enter the name of the file to be assembled in the editable combo box as shown in Figure 1-5.

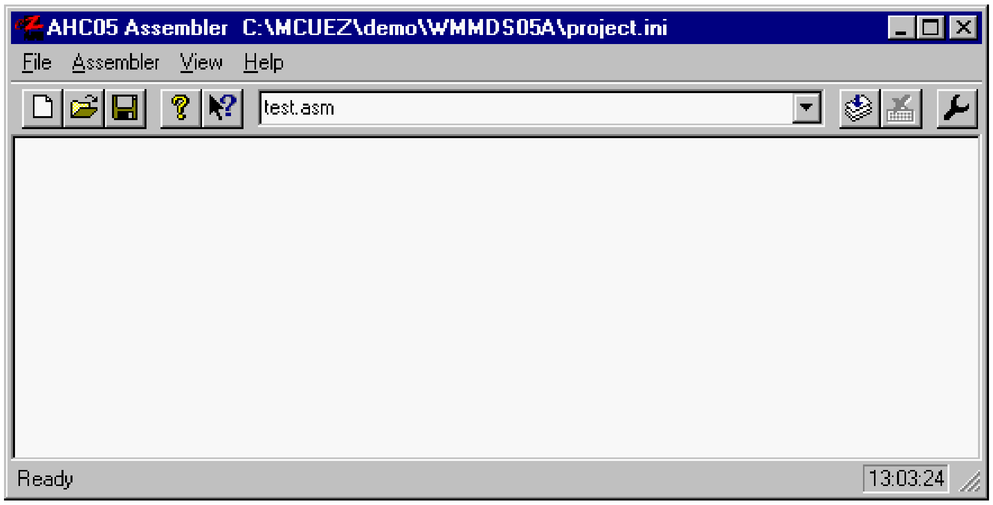
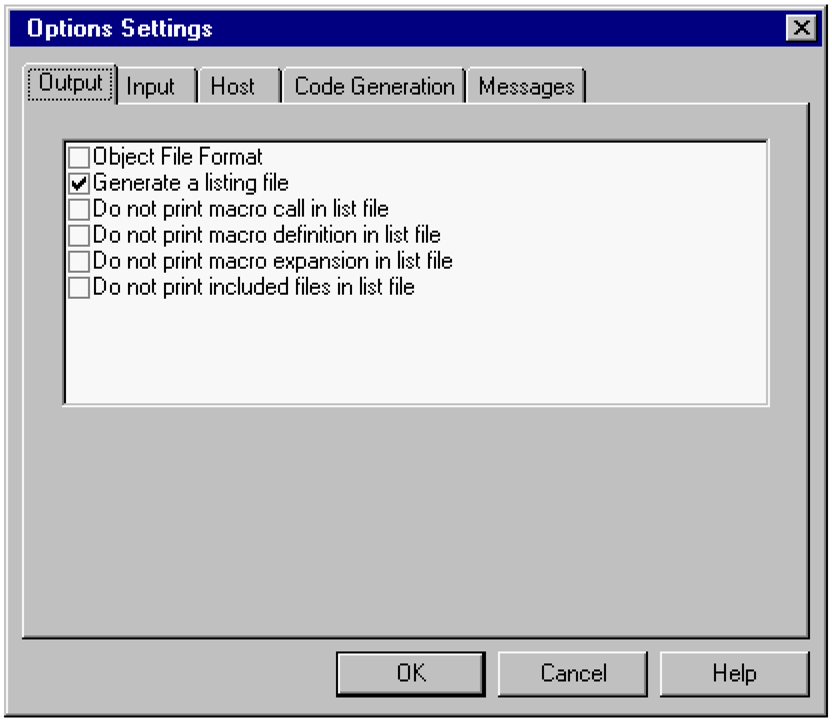


Figure 1-5. Assembler Window

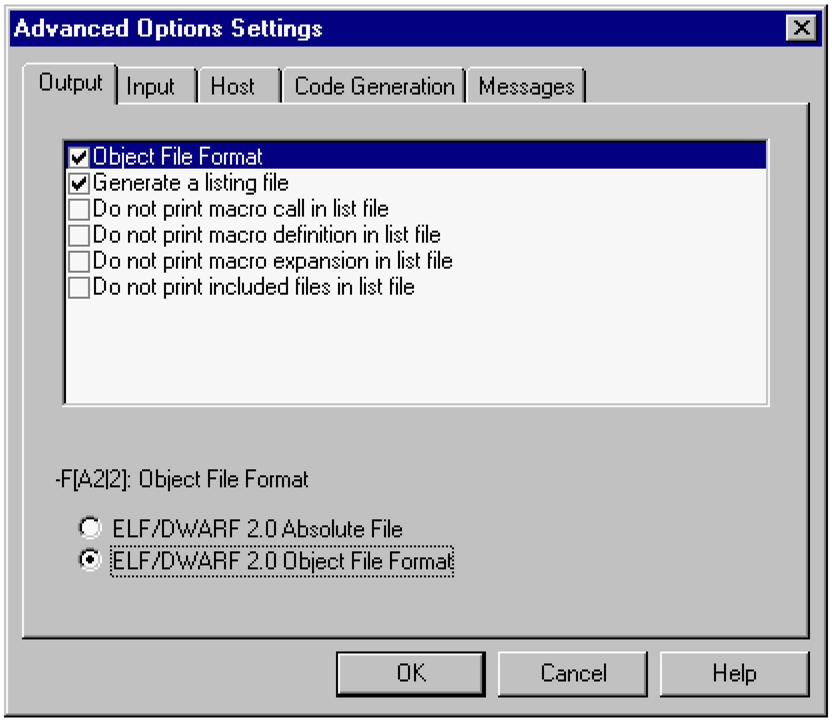
2. To generate an *ELF/DWARF 2.0* object file, the assembler must be set correctly. Select the menu entry **Assembler | Options**.



- [ ] Object File Format
- [✓] Generate a listing file
- [ ] Do not print macro call in list file
- [ ] Do not print macro definition in list file
- [ ] Do not print macro expansion in list file
- [ ] Do not print included files in list file

Figure 1-6. Options Settings Dialog Box

3. In the **Output** tab, select the check box in front of the label **Object File Format** (shown in **Figure 1-7**). Select the radio button **ELF/DWARF 2.0 Object File Format** and click **OK**. The assembler is ready to generate an *ELF/DWARF 2.0* object file.



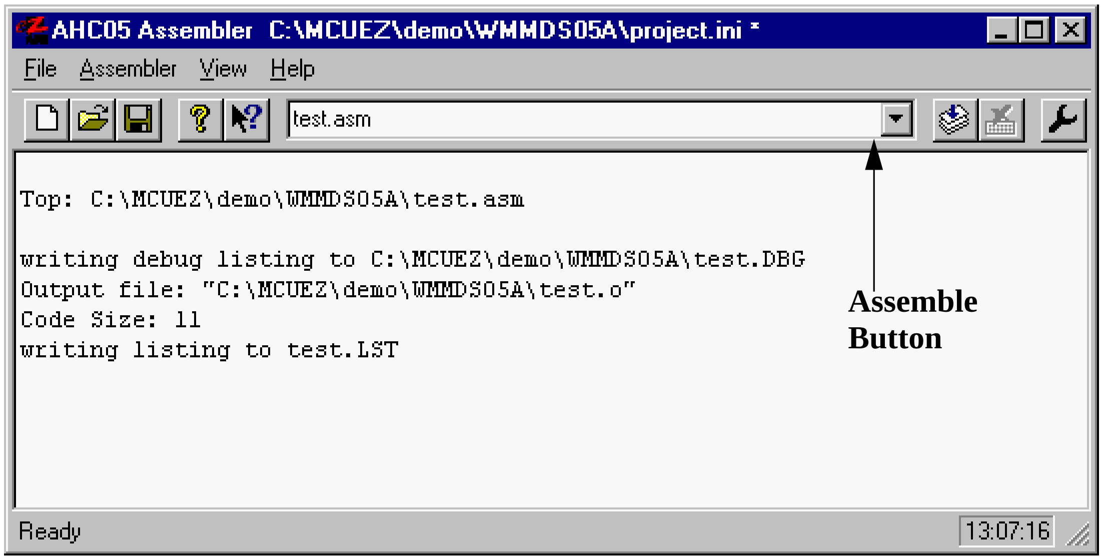
[✓] Object File Format
[✓] Generate a listing file
[ ] Do not print macro call in list file
[ ] Do not print macro definition in list file
[ ] Do not print macro expansion in list file
[ ] Do not print included files in list file

-F[A2|2]: Object File Format

ELF/DWARF 2.0 Absolute File [✓ ELF/DWARF 2.0 Object File Format]

Figure 1-7. Selecting an Object File Format

4. Click on the **Assemble** button to assemble the file. See **Figure 1-8**.



```
Top: C:\MCUEZ\demo\WMMDS05A\test.asm

writing debug listing to C:\MCUEZ\demo\WMMDS05A\test.DBG
Output file: "C:\MCUEZ\demo\WMMDS05A\test.o"
Code Size: 11
writing listing to test.LST
```

Figure 1-8. Assembling a File

The assembler indicates success by printing the number of code bytes generated. The message Code Size: 11 indicates that *test.asm* was assembled without errors. The assembler generates a binary object file and debug listing file for each source file. The binary object file has the same name as the input module with an extension of *.o*. The debug listing file has the same name as the input module with an extension of *.dbg*.

When the assembly option -L is specified on the command line, the assembler generates a list file containing the source instruction and corresponding hexadecimal code.

The list file generated by the assembler looks like this:

```
Motorola HC05-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997
Abs. Rel.   Loc.   Obj. code   Source line
---- ----   ------ ---------   -----------
1    1                              XDEF entry  ; Make the
2    2                                          ; This is
3    3                                          ; symbol and
4    4                                          ; application.
5    5                       cstSec:SECTION     ; Define a
6    6     000000 05         var1:   DC.B 5     ; Assign 5 to
7    7                       dataSec:SECTION    ; Define a
8    8     000000            data:   DS.B 1     ; Define one
9    9                       codeSec:SECTION    ; Define a
10   10                      entry:
11   11    000000 9C                 RSP        ; Reset stack
12   12
13   13    000001 C6 xxxx            LDA   var1
14   14                       main:
15   15    000004 4C                 INCA
16   16    000005 C7 xxxx            STA   data
17   17    000008 20FA               BRA   main
```

1.4.4 Linking an Application

Once the object file is available, link the application. The linker will organize the code and data sections according to the linker parameter file. Follow this procedure to link an application:

1. Start the editor and create the linker parameter file. Copy the provided demo file *fibo.prm* to *test.prm*.
2. In the file *test.prm*, change the name of the executable and object files to *test*.
3. Modify the start and end addresses for the ROM and RAM memory areas.

The module *test.prm* appears as follows:

```
LINK test.abs     /* Name of executable file generated.*/
NAMES test.o END  /* Name of the object files in the application */
SEGMENTS
  MY_ROM  = READ_ONLY  0xB00 TO 0xBFF; /* READ_ONLY memory area */
  MY_RAM  = READ_WRITE 0x080 TO 0x0EF; /* READ_WRITE memory area */
 MY_STK  = READ_WRITE 0x0F0 TO 0x0FF; /* READ_WRITE memory area. */
END
PLACEMENT
  .data    INTO MY_RAM; /*Variables should be allocated in MY_RAM */
  .text    INTO MY_ROM; /* Code should be allocated in MY_ROM */
 .stack   INTO MY_STK; /* Stack will be allocated in MY_STK. */
END
INIT   entry  /* entry point to the application */
VECTOR ADDRESS 0xFFFE entry /* initialization for reset vector */
```

NOTE: *Commands in the linker parameter file are described in detail in the MCUez Linker User's Manual, Motorola document order number MCUEZLNK/D.*

4. Click the **ezLink** button in the **MCUez Shell** to start the linker. In the editable combo box, enter the name of the file to be linked.
5. To start linking, press **Enter** or the **Link** button.

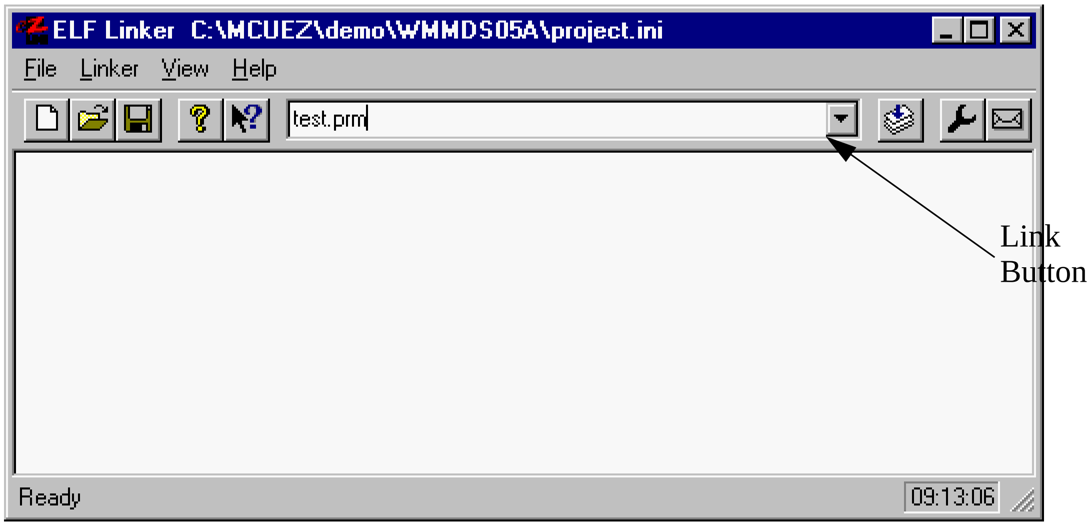
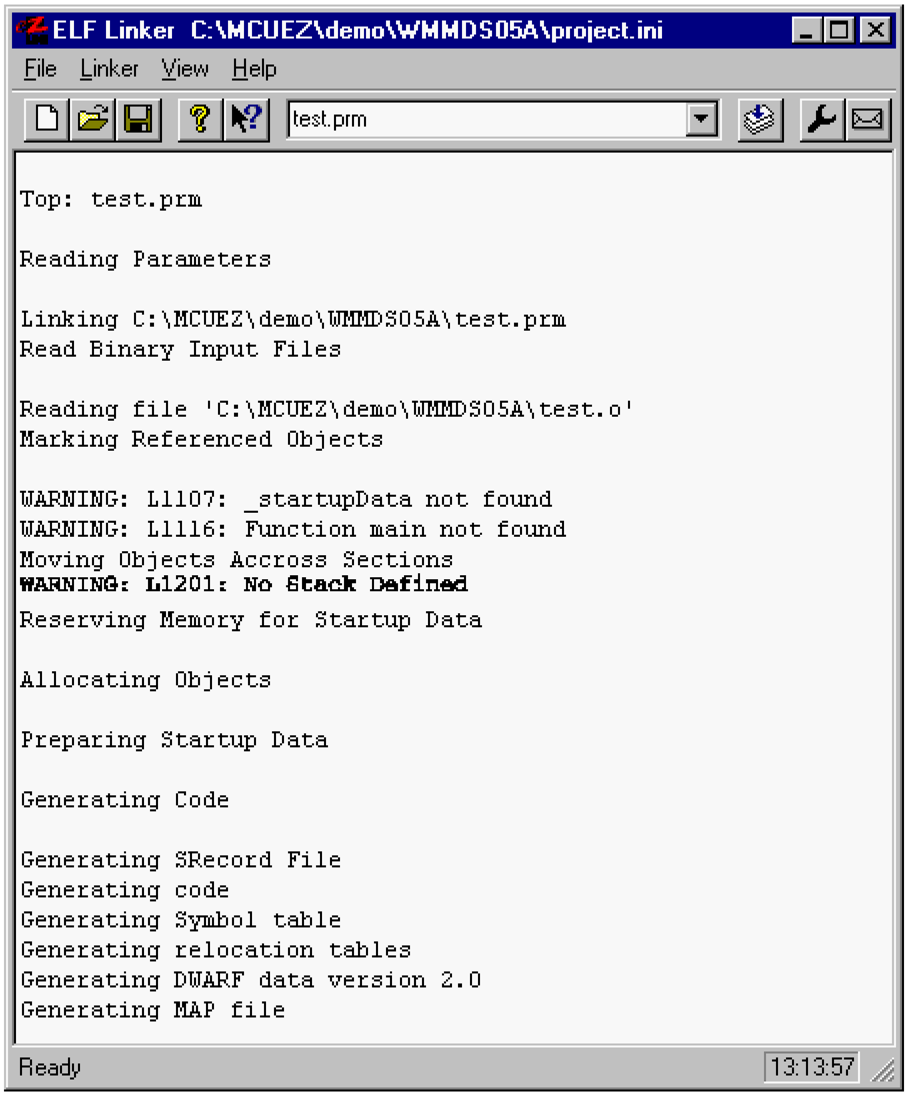


Figure 1-9. Linker Window

The linker window displays the link process as shown in also **Figure 1-10**.



```
Top: test.prm

Reading Parameters

Linking C:\MCUEZ\demo\WMMDS05A\test.prm
Read Binary Input Files

Reading file 'C:\MCUEZ\demo\WMMDS05A\test.o'
Marking Referenced Objects

WARNING: L1107: _startupData not found
WARNING: L1116: Function main not found
Moving Objects Accross Sections
WARNING: L1201: No Stack Defined
Reserving Memory for Startup Data

Allocating Objects

Preparing Startup Data

Generating Code

Generating SRecord File
Generating code
Generating Symbol table
Generating relocation tables
Generating DWARF data version 2.0
Generating MAP file
```

Figure 1-10. Link Process

Section 2. Graphical User Interface

2.1 Contents

2.2 Introduction 37

2.3 Starting the Motorola Assembler 38

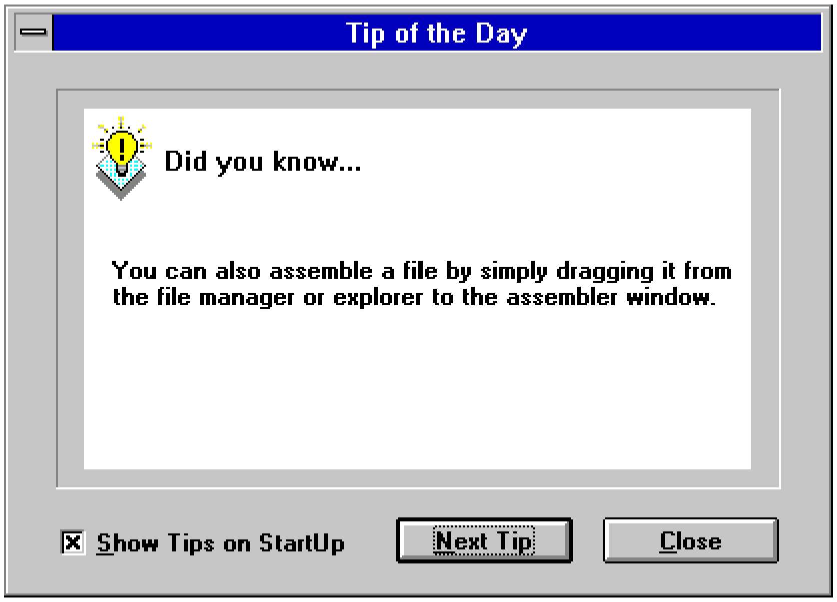
2.4 Assembler Graphical Interface 39
2.4.1 Window Title 39
2.4.2 Content Area 40
2.4.3 Assembler Toolbar 41
2.4.4 Status Bar 42
2.4.5 Assembler Menu Bar 42
2.4.5.1 File Menu 43
2.4.5.1.1 Editor Settings Dialog 44
2.4.5.1.2 Save Configuration Dialog 49
2.4.5.2 Assembler Menu 50
2.4.5.3 View Menu 50
2.4.5.4 Option Settings Dialog Box 51
2.4.6 Specifying the Input File 52
2.4.6.1 Using the Editable Combo Box in the Toolbar 52
2.4.6.2 Using the Entry File | Assembly 53
2.4.6.3 Using Drag and Drop 53
2.4.7 Error Feedback 53

2.2 Introduction

The MCUez HC05 assembler uses a Microsoft Windows® application, which is a graphical user interface (GUI).

2.3 Starting the Motorola Assembler

Start the assembler by clicking on the **ezASM** icon in the **MCUez Shell**. When the assembler is started, a standard **Tip of the Day** window is displayed.



You can also assemble a file by simply dragging it from the file manager or explorer to the assembler window.

Figure 2-1. Tip of the Day Window

Click **Next Tip** to display more information about the assembler.

To disable this window when the assembler is started, uncheck **Show Tips on StartUp**.

To re-enable the tips window, choose **Help|Tip of the Day ...** and check **Show Tips on StartUp.**

2.4 Assembler Graphical Interface

Figure 2-2 depicts the main assembler window.

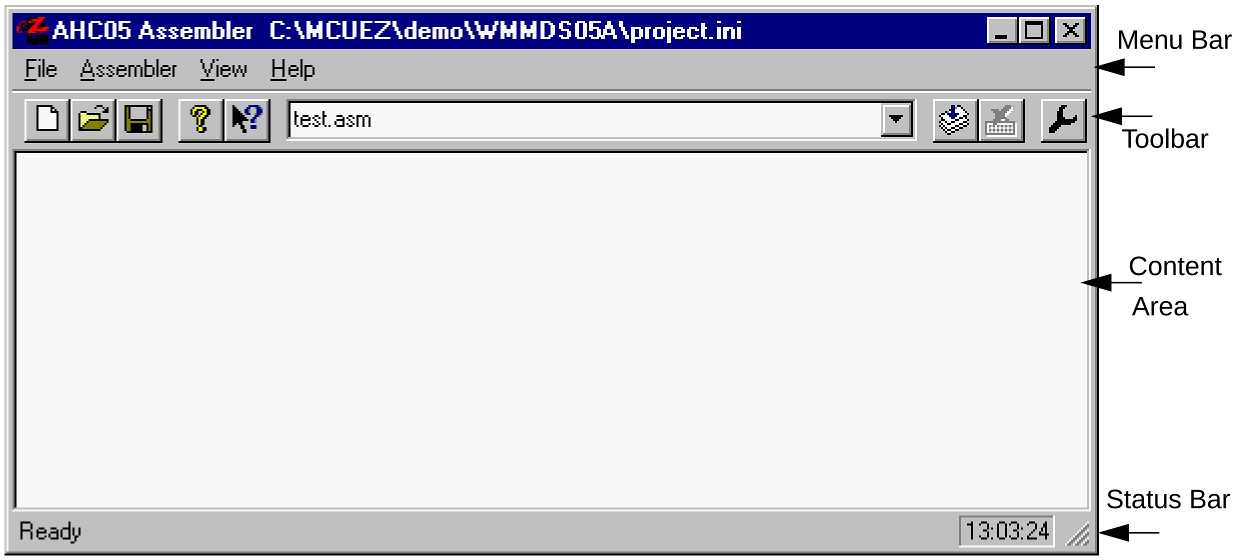


Figure 2-2. Assembler Window

This window is visible only if a filename is not specified while starting the assembler.

The assembler window provides a window title, menu bar, toolbar, content area, and status bar.

2.4.1 Window Title

The window title displays the assembler name and project directory. If no project is loaded, **Default Configuration** is displayed. An asterisk (*) after the configuration name indicates that some values have changed. This could be changes in options, editor configuration, or window appearance (position, size, font, etc.).

2.4.2 Content Area

The content area displays logging information about the assembly session. Logging information consists of:

- Name of file being assembled
- Complete path and name of files processed (main assembly file and included files)
- List of error, warning, and information messages
- Size of code generated during assembly

When a filename is dragged and dropped into the content area, the file is either assembled or loaded as a configuration file. If the file has a *.ini* extension, it is loaded as a new configuration file. If not, the file is assembled with the current option settings.

Assembly information in the content area includes:

- Files created or modified
- Location within file where errors occurred
- A message number

Some files listed in the content area can be opened in the editor specified during project configuration. Double click on a filename to open an editable file or select a line that contains a filename and click the right mouse button to display a menu that contains an **Open ...** entry (if file is editable).

A message number is displayed with message output. From this output, there are three ways to open the corresponding help information.

1. Select one line of the message and press F1. Help for the associated message number is displayed. If the selected line does not have a message number, the main help is displayed.
2. Press Shift-F1 and then click on the message text. If there is no associated message number, the main help is displayed.
3. Click the right mouse button on the message text and select **Help on ...**.

After an assembly session has completed, error feedback can be performed automatically by double clicking on the message in the content area. The source file containing the error or warning message will open to the line containing the problem.

2.4.3 Assembler Toolbar

Figure 2-3 illustrates the assembler toolbar.

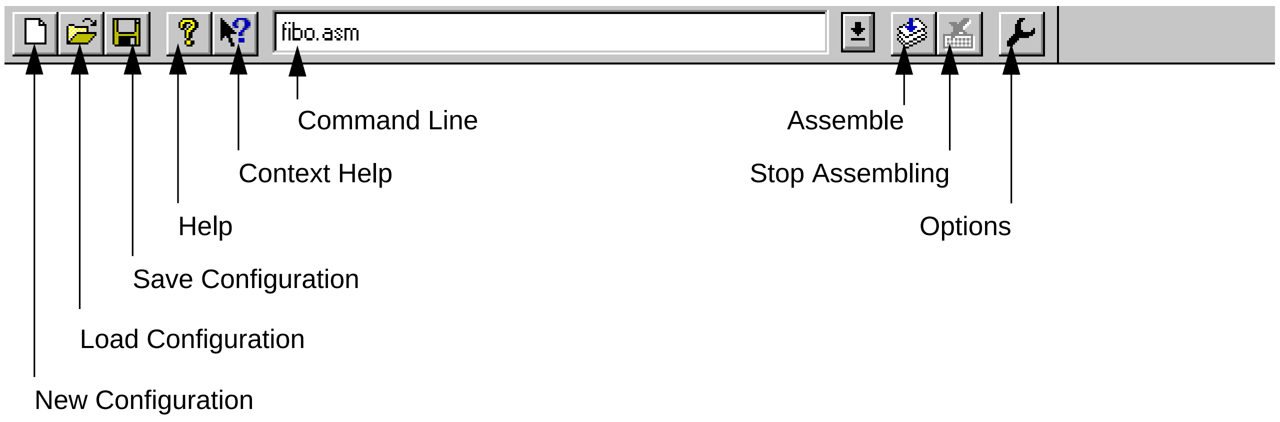


Figure 2-3. Assembler Toolbar

The three buttons on the left correspond with entries in the **File** menu. The **New Configuration**, **Load Configuration,** and **Save Configuration** buttons enable the user to reset, load, or save configuration files.

The **Help** button will open the assembler help file. The **Context Help** button changes the mouse pointer to a question mark beside an arrow. Then click on an item within the application to display help information. Help is available for menus, toolbar buttons, and window areas.

The command line box contains a drop down list of the last commands executed. Once a command line has been selected or entered in the combo box, click the **Assemble** button to execute the command.

The **Options Setting** button opens the **Options Setting** dialog box.

2.4.4 Status Bar

Figure 2-4 shows the assembler status bar.

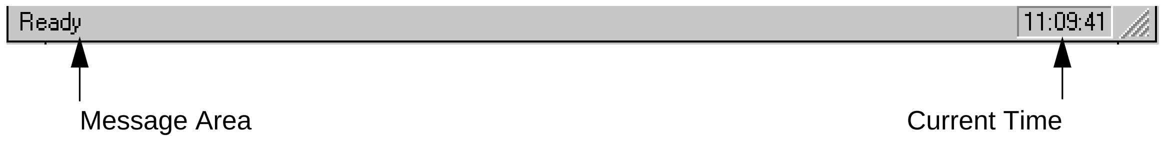


Figure 2-4. Assembler Status Bar

Point to a menu entry or button in the toolbar to display a brief explanation in the message area.

2.4.5 Assembler Menu Bar

The entries in Table 2-1 are available in the menu bar:

Table 2-1. Menu Bar

Menu Entry	Description
File	Assembler configuration file management
Assembler	Assembler option settings
View	Assembler window settings
Help	Standard windows help menu

2.4.6 File Menu

An assembler configuration file typically contains the following information:

- Assembler option settings specified in the assembler dialog boxes
- Last command line executed and current command line
- Window position, size, and font
- Editor associated with the assembler
- **Tip of the Day** settings

Assembler configuration information is stored in the specified configuration file. As many configuration files as required for a project can be defined. Switch to different configuration files by selecting **File|Load Configuration** and **File|Save Configuration** or by clicking the corresponding toolbar buttons.

For instance:

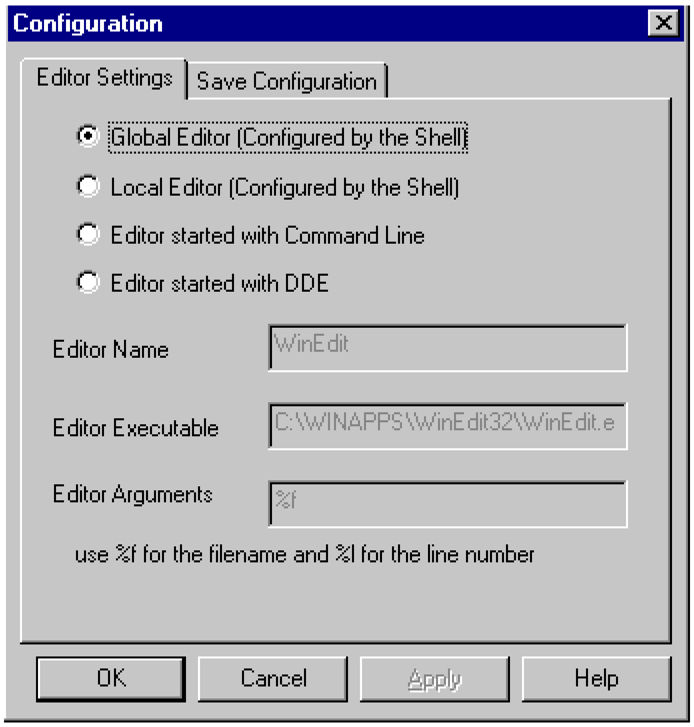
- Choose **File|Assemble** to open a standard **Open File** dialog box. A list of all *.asm* files in the project directory is displayed. Select an input file. Click **OK** to close the dialog box and assemble the selected file.
- Choose **File|New/Default Configuration** to reset assembler options to the default values. Default values are specified in the section titled **Command Line Options.**
- Choose **File|Load Configuration** to open a standard **Open File** dialog box. A list of all *.ini* files in the project directory is displayed. Select a configuration file to be used by subsequent assembly sessions.
- Choose **File|Save Configuration** to store the current settings in the configuration file specified in the title bar.
- Choose **File|Save Configuration as ...** to open a standard **Save As** dialog box and display the list of all *.ini* files in the project directory. Specify the name and location of the configuration file. Click **OK** to save the current settings in the specified configuration file.
- Choose **File|Configuration ...** to open the **Configuration** dialog box. Specify an editor and related information to be used for error feedback, then save the configuration.

2.4.6.1 Editor Settings Dialog

This dialog box has several radio buttons for selecting a type of editor. Depending on the type selected, the content below it changes.

These are the main entries:

- **Global Editor (Configured by the Shell)**



Configuration

Editor Settings | Save Configuration

[● Global Editor (Configured by the Shell)]

○ Local Editor (Configured by the Shell)

○ Editor started with Command Line

○ Editor started with DDE

Editor Name: WinEdit

Editor Executable: C:\WINAPPS\WinEdit32\WinEdit.e

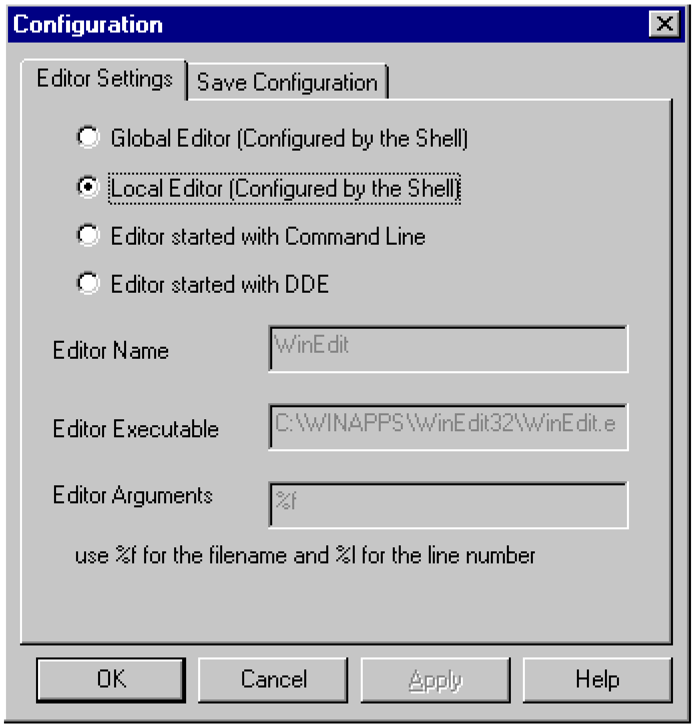
Editor Arguments: %f

use %f for the filename and %l for the line number

Figure 2-5. Starting the Global Editor

This entry is enabled only when an editor is defined in the **[Editor]** section of the global initialization file *mcutools.ini*.

- **Local Editor (Configured by the Shell)**



( ) Global Editor (Configured by the Shell)
(●) Local Editor (Configured by the Shell)
( ) Editor started with Command Line
( ) Editor started with DDE

Editor Name: WinEdit
Editor Executable: C:\WINAPPS\WinEdit32\WinEdit.e
Editor Arguments: %f

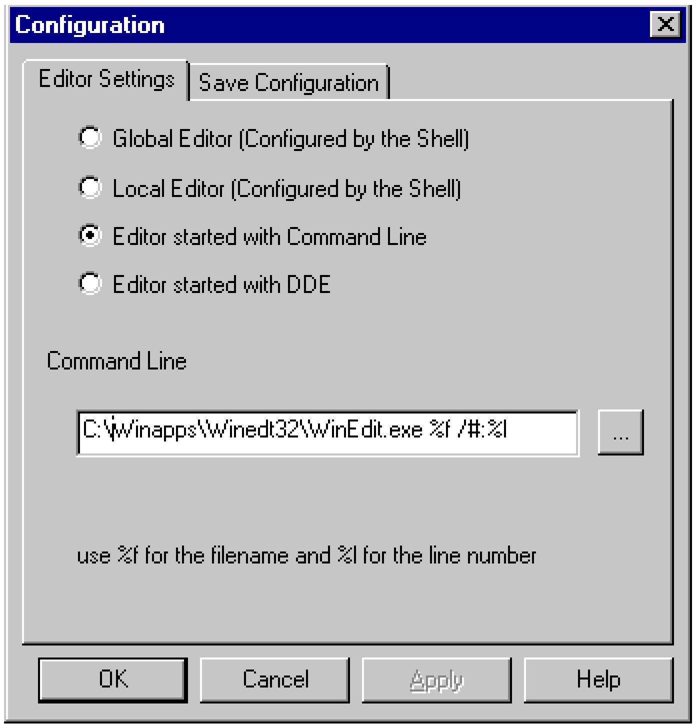
use %f for the filename and %l for the line number

Figure 2-6. Starting the Local Editor

This entry is enabled only if an editor is defined in the local configuration file, usually *project.ini* in the project directory.

The **Global Editor** and **Local Editor** settings cannot be edited within this dialog box, since they are read only. These entries can be configured with the **MCUez Shell** application.

- **Editor started with Command Line**



Global Editor (Configured by the Shell)

Local Editor (Configured by the Shell)

[✓ Editor started with Command Line]

Editor started with DDE

Command Line

```
C:\Winapps\Winedt32\WinEdit.exe %f /#:%l
```

use %f for the filename and %l for the line number

Figure 2-7. Starting the Editor with the Command Line

When this editor type is selected, a separate editor is associated with the assembler for error feedback. The editor configured in the shell will not be used for error feedback. Enter the appropriate path and command name to start the editor. Command modifiers are specified on the command line. For example:

For WinEdit™ 32-bit version

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

For Write

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

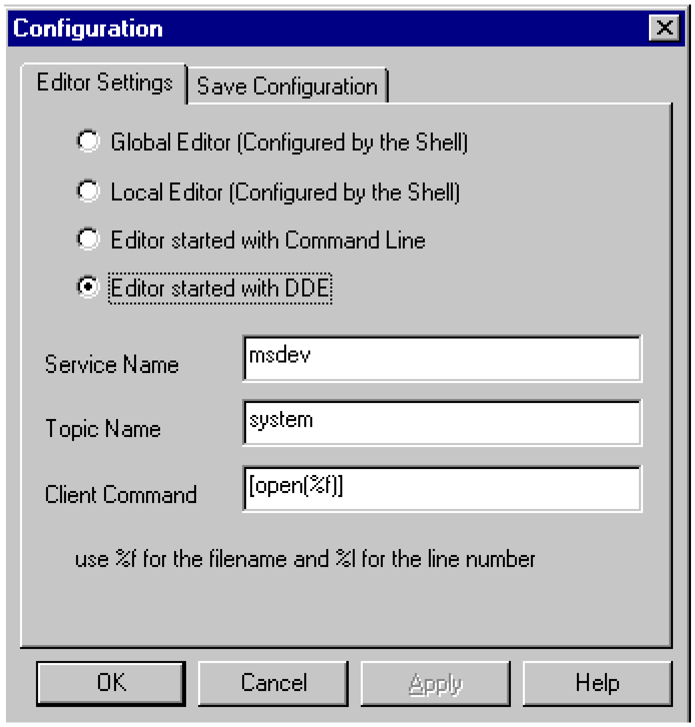
Write does not support line number modifier.

For Motpad

```
C:\TOOLS\MOTPAD\MOTPAD.exe %f::%l
```

Motpad supports line numbers.

- **Editor started with DDE**



( ) Global Editor (Configured by the Shell)
( ) Local Editor (Configured by the Shell)
( ) Editor started with Command Line
(●) Editor started with DDE

Service Name	msdev
Topic Name	system
Client Command	[open(%f)]

use %f for the filename and %l for the line number

Figure 2-8. Starting the Editor 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 filename and line number as explained in the next example.

Example:

For Microsoft Developer Studio®, use this setting:

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

- Modifiers

  When either entry **Editor Started with the Command line** or **Editor started with DDE** is selected, the configuration may contain modifiers to identify which file to open and which line to select.

  – The `%f` modifier refers to the name of the file (including path) where the error has been detected.

  – The `%l` modifier refers to the line number where the message has been detected.

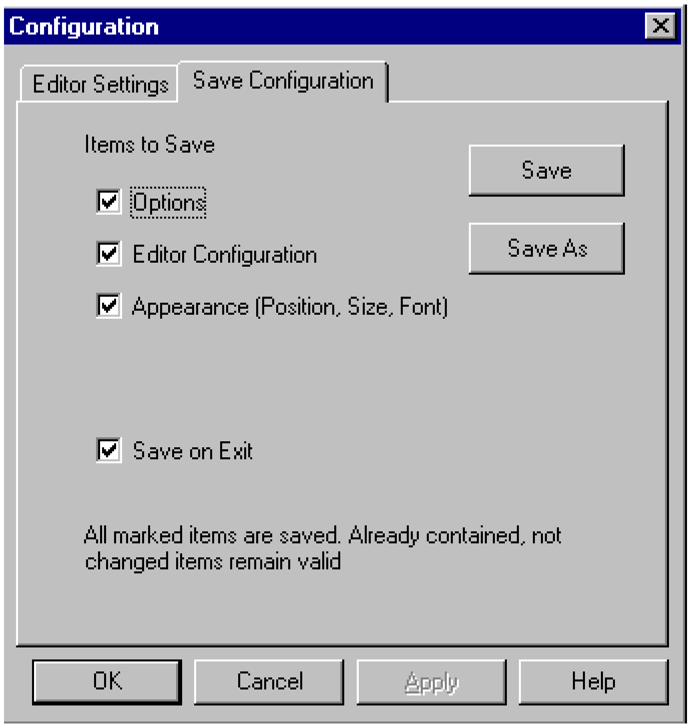
The editor format depends on the command syntax used to start the editor. Check the editor manual for modifiers that can be used to define the editor command line.

CAUTION: *Be cautious when using the `%l` modifier. This modifier can be used only with an editor that can be started with a line number as a parameter. Editors such as WinEdit version 3.1 or lower and Notepad do not allow this kind of parameter.*

NOTE: *When using a word processing editor, such as Microsoft Word® or Wordpad, make sure to save the input file as an ASCII text file; otherwise, the assembler will have trouble processing the file.*

2.4.6.2 Save Configuration Dialog

Figure 2-9 shows the **Save Configuration** dialog box.



Items to Save

[✓] Options

[✓] Editor Configuration

[✓] Appearance (Position, Size, Font)

[✓] Save on Exit

All marked items are saved. Already contained, not changed items remain valid

Figure 2-9. Save Configuration Dialog Box

The second page of the configuration dialog consists of save operations. In the **Save Configuration** dialog, select attributes to be stored in the project file. This dialog box provides these configurations:

- **Options** — When set, the current option settings are stored in the configuration file. Disable this option to retain the last saved options.
- **Editor configuration** — When set, the current editor settings are stored in the configuration file. Disable this option to retain the last saved options.
- **Appearance** — When set, the current application appearance, such as the window position (only loaded at startup time) and the command line content and history, is saved. Disable to keep previous settings.

- **Save on exit** — If this option is set, the assembler will save the configuration on exit. No prompt will appear to confirm this operation. If this option is not set, the assembler will ignore any changes.

NOTE: *Almost all settings are stored in the configuration file. Exceptions are the recently used configuration list and all settings in this dialog. These settings are stored in the assembler section of the mcutools.ini file.*

Assembler configurations can coexist in the same file used for the project configuration (defined by the shell application) along with other MCUez tool specifications. When an editor is configured by the shell, the assembler can read this information from the project file, if present. The project configuration file created by the shell is named project.ini. Therefore, this filename is also suggested (but not mandatory) to the assembler.

2.4.7 Assembler Menu

Table 2-2 depicts the **Assembler** menu that allows customization of the assembler and setting or resetting of assembler options.

Table 2-2. Assembler Menu

Item	Description
Options	Allows defining of the options to be activated when assembling an input file

2.4.8 View Menu

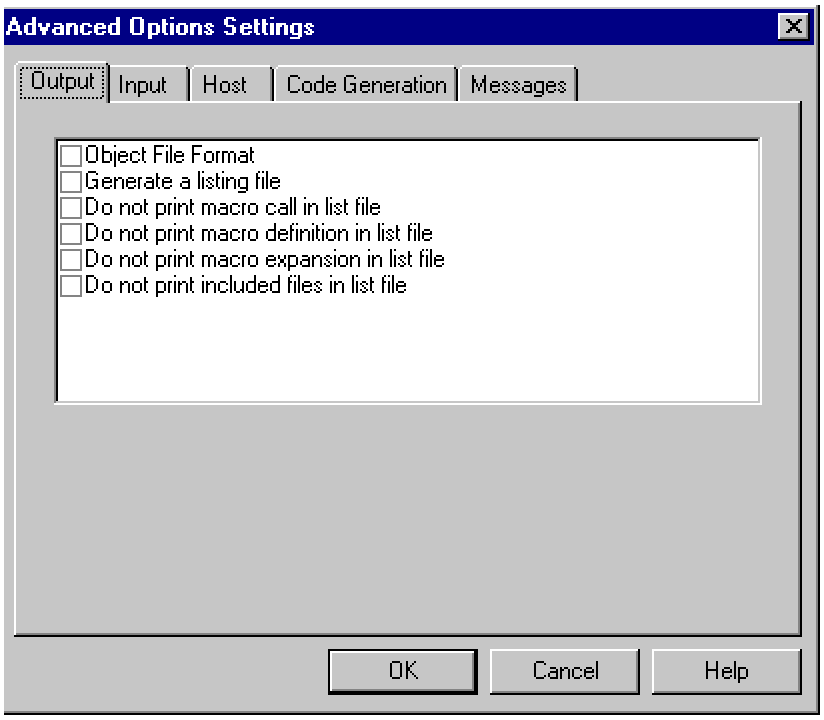
This menu enables customization of the assembler window. For instance, whether the status bar or toolbar will be displayed or hidden can be defined. The user also can define the font used in the window or can clear the window.

- Choose **View|Tool Bar** to switch on/off the assembler window toolbar.
- Choose **View|Status Bar** to switch on/off the assembler window status bar.
- Choose **View|Log ...** to customize the output in the assembler window content area.

- Choose **View|Log ...|Change Font** to open a standard **Font Selection** dialog box. Options selected in this dialog are applied to the assembler window content area.
- Choose **View|Log ...|Clear Log** to clear the assembler window content area.

2.4.8.1 Option Settings Dialog Box

This dialog box enables the user to set/reset assembler options, as shown in **Figure 2-10**.



☐ Object File Format
☐ Generate a listing file
☐ Do not print macro call in list file
☐ Do not print macro definition in list file
☐ Do not print macro expansion in list file
☐ Do not print included files in list file

Figure 2-10. Option Settings Dialog Box

Available options are arranged in different groups as shown in Table 2-3.

Table 2-3. Advanced Options

Option Group	Description
Output	Lists options related to the output files generated (type of files to be generated)
Input	Lists options related to the input file
Host	Lists options related to the host
Code generation	Lists options related to code generation (memory models, etc.)
Messages	Lists options controlling the generation of error messages

An assembly option is set when the corresponding check box is checked. To obtain more information about a specific option, select the option and press the F1 key or the **Help** button. To select an option, click once on the option text.

NOTE: *Options that require additional parameters will display an edit box or an additional subwindow where additional parameters can be set.*

Assembler options specified in the project file (using the **MCUez Shell**) are displayed automatically in the **Option Settings** dialog box.

2.4.9 Specifying the Input File

The input file to be assembled can be specified in several ways. During the assembly session, options will be set according to the configuration provided by the user in the **Option Settings** dialog box. Before assembling a file, make sure a project directory is associated with the assembler.

2.4.9.1 Using the Editable Combo Box in the Toolbar

The following describes how to use the **Editable Combo** box.

- **Assembling a new file** — A new filename and additional assembler options can be entered on the command line. Click on the assemble button or press the **Enter** key to assemble the specified file.

- Re-assembling a file — The previously executed command can be displayed by clicking on the arrow on the right side of the command line. From the drop down list, select a command. Click on the **Assemble** button or press the **Enter** key to assemble the specified file.

2.4.9.2 Using the Entry File | Assembly ...

Select the menu entry **File | Assemble** to display the **File to Assemble** dialog box. Browse to and select the desired file. Click **Open** to assemble the selected file.

2.4.9.3 Using Drag and Drop

A filename can be dragged from an external program (for example, **File Manager**) and dropped into the assembler window. The dropped file is assembled as soon as the mouse button is released in the assembler window. If the dragged file has the extension *.ini*, it is a configuration file and will be loaded and not assembled.

2.5 Error Feedback

After a source file has been assembled, the content area displays a list of all error or warning messages detected. The message format is:

>> <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 “C:\DEMO\fiboerr.asm”, line 76, col 20, pos 1932
   BRA label
            ^
ERROR A1104: Undeclared user defined symbol: label
```

Errors can be corrected by using the editor defined during configuration. Editors such as WinEdit version 95 (or higher) or Codewright from Premia Corporation can be started with a line number in the command line. If configured correctly, these editors are activated automatically by double clicking on an error message. The editor will open the file containing the error and position the cursor on the line with the error.

Editors like WinEdit version 31 or lower, Notepad, or Wordpad cannot be started with a line number. These editors can be activated automatically by double clicking on a message. The editor will open the file containing the error. To locate the error, use the find or search feature of the editor. In the assembler content area, select the line containing the message class, number, and string and press CTRL+C to copy the message. Paste the message in the **Find** dialog box of the editor to search for the error.

Section 3. Environment Variables

3.1 Contents

3.2 Introduction 56

3.3 Paths 57

3.4 Line Continuation 57

3.5 Environment Variables Description 58
3.5.1 ASMOPTIONS 59
3.5.2 GENPATH 60
3.5.3 ABSPATH 61
3.5.4 OBJPATH 62
3.5.5 TEXTPATH 63
3.5.6 SRECORD 64
3.5.7 ERRORFILE 65
3.5.8 COPYRIGHT: Copyright Entry in Object File 67
3.5.9 INCLUDETIME: Creation Time in Object File 68
3.5.10 USERNAME: User Name in Object File 69

3.2 Introduction

This part of the manual describes the environment variables used by the assembler. Some of the environment variables are also used by other tools, for example the linker.

Various assembler parameters may be set with environment variables.

The syntax is:

KeyName=ParameterDefinition

Example:

```
GENPATH=C:\INSTALL\LIB;D:\PROJECTS\TEST
```

NOTE: *No blank space is allowed before and after the* = *(equal sign) character in the definition of an environment variable.*

These parameters are defined in three ways:

1. Using system environment variables supported by an operating system
2. Putting definitions in a file named *default.env* in the default directory
3. Putting definitions in a file specified in the system environment variable ENVIRONMENT

NOTE: *The default directory can be set with the environment variable DEFAULTDIR.*

When looking for an environment variable, all programs first search the system environment, then the *default.env* file, and finally the global environment file set by ENVIRONMENT. If no definition is found, a default value is assumed.

NOTE: *The environment may also be changed using the -Env assembler option.*

3.3 Paths

Most environment variables contain path lists indicating where to look for files. If a directory name is preceded by an asterisk (*), the programs recursively search the subdirectories of the specified directory. The directories are searched in the order they appear in the path list. A path list is a list of directory names separated by semicolons following this syntax:

PathList=DirSpec;DirSpec

DirSpec=*DirectoryName

Example:

```
GENPATH=C:\INSTALL\LIB;D:\PROJECTS\TESTS
```

Example:

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

The preferred method is to use the **MCUez Shell** to define the environment by means of a *default.env* file in the project directory. Separate projects can exist in separate directories, each with its own environment file.

For some environment variables, a synonym also exists. Synonyms may be used for compatibility with older releases of the assembler.

3.4 Line Continuation

It is possible to specify an environment variable over more than one line by using the line continuation character \ (back slash):

Example:

```
ASMOPTIONS=\
-W2 \
-WmsgNe=10
```

This is the same as:

```
ASMOPTIONS=-W2 -WmsgNe=10
```

Observe the following when using the continuation character in path definitions.

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

Will result in:

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

To avoid syntax errors, use a semicolon (;) at the end of a path if there is a \ at the end of the code line, such as:

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

3.5 Environment Variables Description

The remainder of this section describes each of the environment variables available for the assembler, including **Table 3-1**.

Table 3-1. Environment Variables

Topic	Description
Syntax	Syntax of option in EBNF (Extended Backus-Naur Form) format
Arguments	Describes and lists optional and required arguments for the
Default	Shows the default setting for the variable, if applicable
Description	Provides a detailed description of the environment variable and how to use it
Example	Gives an example of usage and effects of the variable where possible. The examples show an entry in the *default.env* file for a PC or in the *.hidefaults* file for UNIX®.
Tools	Lists tools that use this variable, if applicable
MCUez Shell	Explains how the environment variable can be initialized in the **MCUez Shell**
See also	Lists related sections, if applicable

3.5.1 ASMOPTIONS

Syntax:	`ASMOPTIONS={<option>}`
Arguments:	<option>: Assembler command line option
Description:	If this environment variable is set, the assembler appends the options to its command line each time a file is assembled. It can be used to globally specify certain options that should always be set, so they don't have to be specified each time a file is assembled. Options listed here must be valid assembler options and are separated by space characters.
Example:	`ASMOPTIONS=-W2 -L`
MCUez Shell:	Open the **Current Configuration** dialog box. Select the **Additional** tab. Enter the environment variable definition in the edit box.
See also:	**Section 5. Assembler Options**

3.5.2 GENPATH

Syntax:	`GENPATH={<path>}`
Arguments:	<path>: Paths separated by semicolons, without spaces
Description:	The macro assembler will look for the source or included 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 entire directory tree is searched recursively, for example, all subdirectories are searched.*

Example:	`GENPATH=\sources\include;...\...\headers;`
MCUez Shell:	Open the **Current Configuration** dialog. Select the **Paths** tab. In the **Configure** combo box, select **General Path**. Enter the directories in the list box (one directory on each line).
See also:	None

3.5.3 ABSPATH

Syntax:	`ABSPATH={<path>}`
Arguments:	<path>: Paths separated by semicolons, without spaces
Description:	This environment variable is only relevant when absolute files are directly generated by the macro assembler instead of object files. When this environment variable is defined, the assembler will store the absolute files it produces in the first directory specified. If ABSPATH is not set, the generated absolute files will be stored in the directory where the source file was found.
Example:	`ABSPATH=\sources\bin;..\..\headers;`
MCUez Shell:	Open the **Current Configuration** dialog. Select the **Paths** tab. In the **Configure** combo box, select **Absolute**. Enter the directories in the list box (one directory on each line).
See also:	None

3.5.4 OBJPATH

Syntax:	`OBJPATH={<path>}`
Arguments:	<path>: Paths separated by semicolons, without spaces
Description:	When this environment variable is defined, the assembler will store the object files it produces in the first directory specified. If OBJPATH is not set, the generated object files will be stored in the directory where the source file was found.
Example:	`OBJPATH=\sources\bin;..\..\headers;`
MCUez Shell:	Open the **Current Configuration** dialog. Select the **Paths** tab. In the **Configure** combo box, select **Object**. Enter the directories in the list box (one directory on each line).
See also:	None

3.5.5 TEXTPATH

Syntax:	`TEXTPATH={`<path>`}`
Arguments:	<path>: Paths separated by semicolons, without spaces
Description:	When this environment variable is defined, the assembler will store the listing files it produces in the first directory specified. If TEXTPATH is not set, the generated listing files will be stored in the directory where the source file was found.
Example:	`TEXTPATH=\sources\txt;..\..\headers;`
MCUez Shell	Open the **Current Configuration** dialog. Select the **Paths** tab. In the **Configure** combo box, select **Text**. Enter the directories in the list box (one directory on each line).
See also:	None

3.5.6 SRECORD

Syntax: `SRECORD=`<RecordType>

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

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

When the SRECORD variable is not set, the type of S record generated will depend on the size of the target address space that is loaded. If the address space can be coded on two bytes, an S1 record is generated. If the address space is coded on three bytes, an S2 record is generated. Otherwise, an S3 record is generated.

NOTE: *If the environment variable SRECORD is set, it is the user's responsibility to specify the appropriate S record type. If S1 is specified while the code is loaded above 0xFFFF, the Motorola S file generated will not be correct because the addresses will all be truncated to 2-byte values.*

Example: `SRECORD=S2`

MCUez Shell: Open the **Current Configuration** dialog.
Select the **Additional** tab.
Enter the environment variable in the list box.

See also: None

3.5.7 ERRORFILE

Syntax:	`ERRORFILE=`<filename>
Arguments:	<filename>: Filename with possible format specifiers

Description: The environment variable ERRORFILE specifies the name of the error file used by the assembler.

Possible format specifiers are:

%n: Substitute with the filename, without the path.

%p: Substitute with the path of the source file.

%f: Substitute with the full filename (path included; same as %p%n).

Examples:

```
ERRORFILE=MyErrors.err
```

Logs all errors in the file *MyErrors.err* in the current directory.

```
ERRORFILE=\tmp\errors
```

Logs all errors in the file named *errors* in the directory *\tmp*

```
ERRORFILE=%f.err
```

Logs all errors in a file with the same name as the source file (with extension *.err*) into the same directory as the source file. For example, if the file *\sources\test.asm* is assembled, an error file *\sources\test.err* will be generated.

```
ERRORFILE=\dir1\%n.err
```

An error file *\dir1\test.err* will be generated for a source file named *test.asm*.

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

An error file *\dir1\dir2\errors.txt* will be generated for a source file *\dir1\dir2\test.asm*. If the environment variable ERRORFILE is not set, errors are written to the default error file. The default error filename is dependent upon how the assembler is configured and started. If no filename is provided, errors are written to the *err.txt* file in the project directory.

Example: This example shows the use of this variable with the WinEdit editor, which looks for an error file called *EDOUT*.

```
Installation directory: E:\INSTALL\PROG
Project sources: D:\JCUNNING
Common Sources for projects: E:\CLIB

Entry in default.env
(D:\JCUNNING\DEFAULT.ENV):
ERRORFILE=E:\INSTALL\PROG\EDOUT

Entry in WINEDIT.INI (in Windows directory):
OUTPUT=E:\INSTALL\PROG\EDOUT
```

MCUez Shell: Open the **Current Configuration** dialog.
Select the **Additional** tab.
Enter the environment variable definition in the list box.

See also: None

3.5.8 COPYRIGHT: Copyright Entry in Object File

Tools:	Compiler, assembler, linker, librarian
Syntax:	`COPYRIGHT=`<copyright string>
Arguments:	<copyright string>: string for the copyright entry in the object file
Default:	None
Description:	Each object file contains an entry for a copyright string. This information may be retrieved from the object files.
Example:	`COPYRIGHT=Copyright by Motorola`
MCUez Shell:	Open the **Current Configuration** dialog. Select the **Additional** tab. Enter the environment variable definition in the list box.
See also:	**3.5.9 INCLUDETIME: Creation Time in Object File**
	3.5.10 USERNAME: User Name in Object File

3.5.9 INCLUDETIME: Creation Time in Object File

Tools:	Compiler, assembler, linker, librarian
Syntax:	`INCLUDETIME=`(ON | OFF)
Arguments:	ON: Include time information in object file.
	OF: Do not include time information in object file.
Default:	ON
Description:	Normally, each object file created contains a time stamp indicating the creation time and data as strings. So whenever a new file is created by one of the tools, the new file gets a new time stamp entry.
	This behavior may be undesired if a binary file compare has to be performed. Even if the information in two object files is the same, the files do not match exactly because the time stamps are not the same. To avoid such problems, this variable may be set to OFF. In this case, the time stamp strings for date and time are “none” in the object file.
	The time stamp may be retrieved from the object files using a decoder.
Example:	`INCLUDETIME=OFF`
MCUez Shell:	Open the **Current Configuration** dialog. Select the **Additional** tab. Enter the environment variable definition in the list box.
See also:	**3.5.8 COPYRIGHT: Copyright Entry in Object File**
	3.5.10 USERNAME: User Name in Object File

3.5.10 USERNAME: User Name in Object File

Tools:	Compiler, assembler, linker, librarian
Syntax:	`USERNAME=`<user>
Arguments:	<user>: Name of user
Default:	None
Description:	Each object file contains an entry identifying the user who created the object file. This information may be retrieved from the object files using a decoder.
Example:	`USERNAME=MOTOROLA`
MCUez Shell	Open the **Current Configuration** dialog. Select the **Additional** tab. Enter the environment variable definition in the list box.
See also:	**3.5.8 COPYRIGHT: Copyright Entry in Object File**
	3.5.9 INCLUDETIME: Creation Time in Object File

Section 4. Files

4.1 Contents

4.2 Introduction. 71

4.3 Input Files 71
4.3.1 Source Files 72
4.3.2 Include Files. 72

4.4 Output Files. 72
4.4.1 Object Files 72
4.4.2 Absolute Files 73
4.4.3 Motorola S Files. 73
4.4.4 Listing Files 73
4.4.5 Debug Listing Files 74
4.4.6 Error Listing Files 74

4.2 Introduction

This chapter describes all file types associated with the MCUez application.

4.3 Input Files

The assembler accepts two forms of input files:

- Source files
- Include files

4.3.1 Source Files

The macro assembler takes any file as input and does not require the filename to have a special extension. Source files will be searched first in the project directory and then in the GENPATH directory.

4.3.2 Include Files

The search for include files is governed by the environment variable GENPATH. Include files are searched for first in the project directory, then in the directories specified in the environment variable GENPATH.

4.4 Output Files

The assembler produces six types of output files:

1. Object files
2. Absolute files
3. Motorola S files
4. Listing files
5. Debug listing files
6. Error listing files

4.4.1 Object Files

After a successful assembly session, the macro assembler generates an object file containing the target code as well as some debugging information. This file is written to the directory given in the environment variable OBJPATH. If that variable contains more than one path, the object file is written to the first directory given. If this variable is not set, the object file is written to the directory where the source file was found. Object files always get the extension *.o*.

4.4.2 Absolute Files

When an application is encoded in a single module and all the sections are absolute sections, the user can decide to generate an absolute file instead of an object file. This file is written to the directory given in the environment variable ABSPATH. If that variable contains more than one path, the absolute file is written in the first directory given. If this variable is not set, the absolute file is written in the directory where the source file was found. Absolute files always get the extension *.abs*.

4.4.3 Motorola S Files

When an application is encoded in a single module and all the sections are absolute sections, the user can decide to generate an absolute file instead of an object file. In that case, a Motorola S record file is generated at the same time. This file can be burnt into an EPROM. It contains information stored in all READ_ONLY sections in the application. The extension for the generated Motorola S record file depends on the SRECORD variable setting.

For instance:

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

This file is written to the directory given in the environment variable ABSPATH. If that variable contains more than one path, the Motorola S file is written in the first directory given. If this variable is not set, the file is written in the directory where the source file was found.

4.4.4 Listing Files

After a successful assembly session, the macro assembler generates a listing file containing each assembly instruction with its associated hexadecimal code. This file is generated when the option `-L` is activated, even if the macro assembler generates an absolute file. 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 specified. If this variable is not set, the listing file is written in the directory where the source file was found. Listing files always get the extension *.lst*. **Section 10. Assembler Listing File** describes the format of this file.

4.4.5 Debug Listing Files

After a successful assembling session, the macro assembler generates a debug listing file, which will be used to debug the application. This file is always generated, even when the macro assembler generates an absolute file. The debug listing file is a duplicate of the source, where all the macros are expanded and the include files are merged. This file (with the extension *.dbg*) is written to the directory listed in the environment variable OBJPATH. If that variable contains more than one path, the debug listing file is written to the first directory given. If this variable is not set, the file is written in the directory where the source file was found. Debug listing files always get the extension *.dbg*.

4.4.6 Error Listing Files

If the macro assembler detects any errors, it creates an error file. The name and location of this file depend on the settings from the environment variable ERRORFILE.

If the macro assembler's window is open, it displays the full path of all include files read. After successful assembly, the number of code bytes generated and the number of global objects written to the object file are displayed. **Figure 4-1** shows the different structures associated with the assembler.

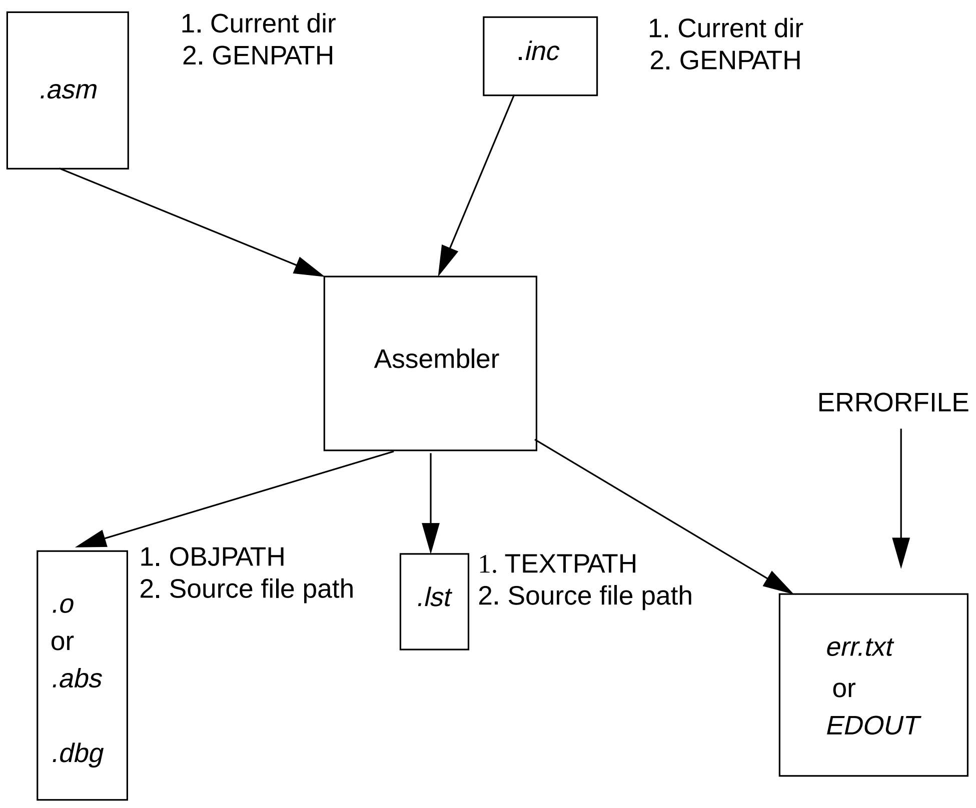


Figure 4-1. Assembler Structural Diagram

Section 5. Assembler Options

5.1 Contents

5.2 Introduction. 78

5.3 ASMOPTIONS 78

5.4 Assembler Options 79
5.4.1 -CI 81
5.4.2 -Env 82
5.4.3 -F2 -FA2 83
5.4.4 -H 84
5.4.5 -L 85
5.4.6 -Lc 87
5.4.7 -Ld 89
5.4.8 -Le 91
5.4.9 -Li. 93
5.4.10 -Ms -Ml 95
5.4.11 -MCUasm. 96
5.4.12 -N 97
5.4.13 -V 98
5.4.14 -W1. 99
5.4.15 -W2. 100
5.4.16 -WmsgNe 101
5.4.17 -WmsgNi 102
5.4.18 -WmsgNw 103
5.4.19 -WmsgFbv -WmsgFbm 104
5.4.20 -WmsgFiv -WmsgFim 106

5.2 Introduction

The assembler offers a number of options to control assembler operation. Options are composed of a dash (`-`) followed by one or more letters or digits. Anything not starting with a dash is assumed to be the name of a source file to be assembled. Assembler options may be specified on the command line or in the ASMOPTIONS environment variable. Typically, each assembler option is specified only once per assembly session.

NOTE: *Arguments for an option must not exceed 128 characters.*

Command line options are not case sensitive. `–Li` is the same as `–li`. For options that belong to the same group, for example `–Lc` and `–Li`, the assembler allows options to be combined, i.e. `–Lci` or `–Lic` instead of `–Lc –Li`.

NOTE: *It is not possible to combine options in different groups, for instance, `–Lc –W1` cannot be abbreviated by the terms `–LC1` or `–LCW1`.*

5.3 ASMOPTIONS

If this environment variable is set, the assembler appends the values (options) defined for this variable to its command line each time a file is assembled. It can be used to globally specify certain options that should always be set, so the user doesn't have to specify them each time a file is assembled.

5.4 Assembler Options

Table 5-1 describes how assembler options are grouped and **Table 5-2** describes the scope of each option.

Table 5-1. Assembler Option Groups

Group	Description
HOST	Lists options related to the host
OUTPUT	Lists options related to output file generation (type of file to be generated)
INPUT	Lists options related to input file
CODE	Lists options related to code generation (memory models, float format, etc.)
MESSAGE	Lists options controlling generation of error messages
VARIOUS	Lists various options

Table 5-2 gives the scope of each option:

Table 5-2. Scope of Assembler Option Groups

Scope	Description
Application	The option has to be set for all files (assembly units) of an application. A typical example is an option to set the memory model. Mixing object files will have unpredictable results.
Assembly unit	The option can be set differently for each assembly unit of an application. Mixing objects in an application is possible.
None	The option is not related to a specific code part. A typical example is options for message management.

Available options are arranged in separate groups, and a dialog box tab is available for each group. The content of the list box depends on the tab selected in the dialog box.

The remainder of this section describes each of the options available for the assembler. The options are listed in alphabetical order and described by the categories in **Table 5-3**.

Table 5-3. Assembler Option Details

Topic	Description
Group	HOST, OUTPUT, INPUT, CODE, MESSAGE, VARIOUS
Scope	Application, assembly unit, or none
Syntax	Specifies the syntax of the option in EBNF format
Arguments	Describes and lists optional and required arguments for the option
Default	Shows the default setting for the option
Description	Provides a detailed description of the option and how to use it
Example	Gives an example of usage and effects of the option where possible. Assembler settings, source code and/or linker PRM (parametric) files are displayed where applicable.
See also	Related options

5.4.1 -CI

-CI:	Set case sensitivity for label names OFF
Group:	INPUT
Scope:	Assembly unit
Syntax:	`-CI`
Arguments:	None
Default:	ON

Description: Switches case sensitivity OFF for label names. When this option is activated, the assembler ignores case sensitivity for label names.

This option can be specified only when the assembler generates an absolute file. (Option -FA2 must be activated.)

Example: When case sensitivity for label names is switched off, the assembler will not generate error messages for this code:

```
   ORG $200
entry: NOP
       BRA Entry
```

The instruction `BRA Entry` will branch on the label `entry`. By default, the assembler is case sensitive for label names. The labels `Entry` and `entry` are two distinct labels.

See also: **Appendix C. P&E Converter**

5.4.2 -Env

-Env:	Set environment variable
Group:	HOST
Scope:	Assembly unit
Syntax:	`-Env` <EnvironmentVariable>=<VariableSetting>
Arguments:	<EnvironmentVariable>: Environment variable to be set <VariableSetting>: Assigned value
Default:	None
Description:	Sets an environment variable
Example:	`ASMOPTIONS=-EnvOBJPATH=\sources\obj` This is the same as `OBJPATH=\sources\obj` in the *default.env* file.
See Also:	**Section 3. Environment Variables**

5.4.3 -F2 -FA2

-F:	Object file format
Group:	OUTPUT
Scope:	Application
Syntax:	`-F (2 | A2)`
Arguments:	2: ELF/DWARF 2.0 object file format
	A2: ELF/DWARF 2.0 absolute file format (default)
Default:	-FA2
Description:	Defines format for the output file generated by the assembler
	With the option -F2 set, the assembler produces an *ELF/DWARF 2.0* object file.
	With the option -FA2 set, the assembler produces an *ELF/DWARF 2.0* absolute file.
Example:	`ASMOPTIONS=-F2`
See also:	None

5.4.4 -H

-H:	Short help
Group:	VARIOUS
Scope:	None
Syntax:	`-H`
Arguments:	None
Default:	None
Description:	The `-H` option will display a short list of available options. No other option or source file should be specified when the `-H` option is invoked.

Example: The following is a portion of the list produced by option `-H`:

```
MESSAGE:
   -N    Show Notification box in case
         of errors
   -W1   Don't print INFORMATION messages
   -W2   Don't print INFORMATION or WARNING
         messages
VARIOUS:
   -H    Prints this list of options
   -V    Prints the Assembler version
```

See also: None

5.4.5 -L

-L:	Generates a listing file
Group:	OUTPUT
Scope:	Assembly unit
Syntax:	-L
Arguments:	None
Default:	None
Description:	Switches on generation of the listing file. This listing file will have the same name as the source file, but with the extension*.lst.* The listing file contains macro definitions, invocation, and expansion lines as well as expanded include files.

Example: `ASMOPTIONS=-L`

In the following assembly code example, the macro cpChar accepts two parameters. The macro copies the value of the first parameter to the second one.

When option `-L` is specified, the following portion of code:

```
         XDEF Start
MyData:  SECTION
char1:   DS.B  1
char2:   DS.B  1
         INCLUDE "macro.inc"
CodeSec: SECTION
Start:
         cpChar char1, char2
         NOP
```

Generates the following output in the assembly listing file:

```
Motorola HC05-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997
Abs. Rel.   Loc.    Obj. code   Source line
---- ----   ------  ---------   -----------
1    1                         XDEF Start
2    2                         MyData: SECTION
3    3   000000                char1:  DS.B  1
4    4   000001                char2:  DS.B  1
5    5                                 INCLUDE "macro.inc"
6    1i                        cpChar: MACRO
7    2i                                   LDA \1
8    3i                                   STA \2
9    4i                                ENDM
10    5i
11    6i
12    6                         CodeSec: SECTION
13    7                         Start:
14    8                             cpChar char1, char2
15    2m   000000 C6 xxxx    +        LDA char1
16    3m   000003 C7 xxxx    +        STA char2
17    9    000006 9D                NOP
18   10    000007 9D                NOP
```

Contents of included files, as well as macro definition, invocation, and expansion are stored in the listing file. Refer to **Section 10. Assembler Listing File** for detailed information.

See also: **5.4.6 -Lc**, **5.4.7 -Ld**, **5.4.8 -Le**, **5.4.9 -Li**

5.4.6 -Lc

-Lc:	No macro call in listing file
Group:	OUTPUT
Scope:	Assembly unit
Syntax:	`-Lc`
Arguments:	None
Default:	None
Description:	Switches on generation of the listing file, but macro invocations are not present in the listing file. The listing file contains macro definitions and expansion lines as well as expanded include files.
Example:	`ASMOPTIONS=-Lc`

In the following assembly code example, the macro cpChar accepts two parameters. The macro copies the value of the first parameter to the second one.

When option `-Lc` is specified, this portion of code:

```
cpChar: MACRO
          LDA \1
          STA \2
        ENDM
codeSec: SECTION
Start:
        cpChar char1, char2
        NOP
        NOP
```

Generates this output in the assembly listing file:

```
Motorola HC05-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997

Abs. Rel.   Loc.   Obj. code   Source line
---- ----   ------ ---------    -----------
1    1
2    2                                  XDEF Start
3    3                         MyData:  SECTION
4    4   000000                char1:   DS.B  1
5    5   000001                char2:   DS.B  1
6    6                                 INCLUDE "macro.inc"
7    1i                        cpChar: MACRO
8    2i                                  LDA \1
9    3i                                  STA \2
10    4i                                ENDM
11    7
12    8                         CodeSec: SECTION
13    9                         Start:
15    2m  000000 C6 xxxx    +          LDA char1
16    3m  000003 C7 xxxx    +          STA char2
17   11   000006 9D                     NOP
18   12   000007 9D                     NOP
19   13   000008 20FE         main:     BRA main
```

Contents of included files, macro definitions, and expansion are stored in the list file. The source line containing the macro call is not present in the listing file. Refer to **Section 10. Assembler Listing File** for detailed information.

See also: **5.4.5 -L**

5.4.7 -Ld

-Ld:	No macro definition in listing file
Group:	OUTPUT
Scope:	Assembly unit
Syntax:	`-Ld`
Arguments:	None
Default:	None
Description:	Switches on generation of the listing file, but macro definitions are not present in the listing file. The listing file contains macro invocation and expansion lines as well as expanded include files.
Example:	`ASMOPTIONS=-Ld`

In the following example, the macro cpChar accepts two parameters. The macro copies the value of the first parameter to the second one. When option `-Ld` is specified, the following portion of code:

```
cpChar:  MACRO
          LDA \1
          STA \2
         ENDM
codeSec: SECTION
Start:
         cpChar char1, char2
         NOP
         NOP
main:    BRA main
```

Generates this output in the assembly listing file:

```
Motorola HC05-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997

Abs. Rel.   Loc.    Obj. code   Source line
---- ----   ------ ---------   -----------
1    1
2    2                                   XDEF Start
3    3                          MyData:  SECTION
4    4    000000                char1:   DS.B  1
5    5    000001                char2:   DS.B  1
6    6
7    7                          cpChar:  MACRO
11   11
12   12                          CodeSec: SECTION
13   13                          Start:
14   14                                  cpChar char1, char2
15     8m   000000 C6 xxxx    +            LDA char1
16     9m   000003 C7 xxxx    +            STA char2
17   15   000006 9D                        NOP
18   16   000007 9D                        NOP
19   17   000008 20FE         main:      BRA main
```

Contents of included files, as well as macro invocation and expansion, are stored in the listing file. Source code from the macro definition is not present in the listing file. Refer to **Section 10. Assembler Listing File** for detailed information.

See also: **5.4.5 -L**

5.4.8 -Le

-Le:	No macro expansion in listing file
Group:	OUTPUT
Scope:	Assembly unit
Syntax:	`-Le`
Arguments:	None
Default:	None
Description:	Switches on generation of the listing file, but macro expansions are not present in the listing file. The listing file contains macro definitions and invocation lines as well as expanded include files.
Example:	`ASMOPTIONS=-Le`

In the following example, the macro cpChar accepts two parameters. The macro copies the value of the first parameter to the second one. When option `-Le` is specified, the following portion of code:

```
cpChar: MACRO
          LDA \1
          STA \2
        ENDM
codeSec: SECTION
Start:
        cpChar char1, char2
        NOP
        NOP
```

Generates this output in the assembly listing file:

```
Motorola HC05-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997

Abs. Rel.   Loc.    Obj. code   Source line
---- ----   ------ ---------   -----------
1    1
2    2                                   XDEF Start
3    3                          MyData:  SECTION
4    4    000000                char1:   DS.B  1
5    5    000001                char2:   DS.B  1
6    6
7    7                          cpChar:  MACRO
11   11
12   12                          CodeSec: SECTION
13   13                          Start:
14   14                                  cpChar char1, char2
17   15    000006 9D                       NOP
18   16    000007 9D                       NOP
19   17    000008 20FE           main:     BRA main
```

Contents of included files, as well as macro definitions and invocation, are stored in the listing file. The macro expansion lines are not present in the listing file. Refer to **Section 10. Assembler Listing File** for detailed information.

See also: **5.4.5 -L**

5.4.9 -Li

-Li:	No included file in listing file
Group:	OUTPUT
Scope:	Assembly unit
Syntax:	`-Li`
Arguments:	None
Default:	None
Description:	Switches on generation of the listing file, but include files are not expanded in the listing file. The listing file contains macro definitions, invocation, and expansion lines.
Example:	`ASMOPTIONS=-Li`

When option `-Li` is specified, the following portion of code:

```
   INCLUDE "macro.inc"
codeSec: SECTION
Start:
        cpChar char1, char2
        NOP
```

Generates this output in the assembly listing file:

```
Motorola HC05-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997

Abs. Rel.   Loc.    Obj. code   Source line
---- ----   ------ ---------   -----------
    1    1
    2    2                              XDEF Start
    3    3                     MyData:  SECTION
    4    4   000000            char1:   DS.B  1
    5    5   000001            char2:   DS.B  1
    6    6                          INCLUDE "macro.inc"
   11    7
   12    8                     CodeSec: SECTION
   13    9                     Start:
   14   10                             cpChar char1, char2
   15   2m   000000 C6 xxxx    +          LDA char1
   16   3m   000003 C7 xxxx    +          STA char2
   17   11   000006 9D                   NOP
   18   12   000007 9D                   NOP
   19   13   000008 20FE       main:     BRA main
```

Macro definition, invocation, and expansion are stored in the listing file.

See also: 5.4.5 -L

5.4.10 -Ms -Ml

-M:	Memory model
Group:	CODE
Scope:	Application
Syntax:	`-M` (s | l)
Arguments:	s: small memory model l: large memory model
Default:	-Ms
Description:	The assembler for the MC68HC05 supports two different memory models. Default is the small memory model, which corresponds to the normal setup, for example, a 64-Kbyte code-address space. If a code memory expansion scheme is used, use the large memory model.
Example:	`ASMOPTIONS=-Ms`
See also:	None

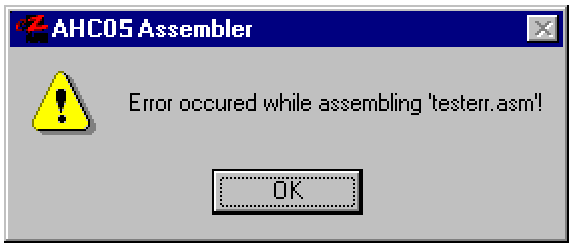
5.4.11 -MCUasm

-MCUasm:	Switch ON MCUasm compatibility
Group:	VARIOUS
Scope:	Assembly unit
Syntax:	`-MCUasm`
Arguments:	None
Default:	None
Description:	Switches ON MCUasm assembler compatibility mode. Additional features supported are listed in **Appendix B. MCUasm Compatibility**.
Example:	`ASMOPTIONS=-MCUasm`
See also:	None

5.4.12 -N

-N:	Displays error notification box
Group:	MESSAGE
Scope:	Function
Syntax:	`-N`
Arguments:	None
Default:	None
Description:	Causes the assembler to display an alert box if an error occurs during assembly. This is useful when running a makefile, since the assembler waits for the user to acknowledge the message, thus suspending makefile processing.
Example:	`ASMOPTIONS=-N`

If an error occurs during assembly, this dialog box is displayed.


AHC05 Assembler

Error occured while assembling 'testerr.asm'!

See also: None

5.4.13 -V

-V:	Displays the assembler version
Group:	VARIOUS
Scope:	None
Syntax:	`-V`
Arguments:	None
Default:	None
Description:	Prints the assembler version and the current directory. This option is useful to determine the current directory.
Example:	`-V` produces the following list:

```
Common Module V-1.0.1, Date Dec 3 1997
                    Directory:   C:\MCUEZ\DEMO\WMMDS05A
                    CCPP User Interface Module, V-1.0.4, Date Jul 10 1997
                    Assembler Target, V-1.0.2, Date Dec 3 1997
```

See also:	None

5.4.14 -W1

-W1:	No information messages
Group:	MESSAGES
Scope:	Assembly unit
Syntax:	`-W1`
Arguments:	None
Default:	None
Description:	INFORMATION messages are not displayed. Only WARNING and ERROR messages are listed.
Example:	`ASMOPTIONS=-W1`
See also:	None

5.4.15 -W2

-W2:	No information and warning messages
Group:	MESSAGES
Scope:	Assembly unit
Syntax:	`-W2`
Arguments:	None
Default:	None
Description:	INFORMATION and WARNING messages are not displayed. Only ERROR messages are listed.
Example:	`ASMOPTIONS=-W2`
See also:	None

5.4.16 -WmsgNe

-WmsgNe:	Number of error messages
Group:	MESSAGES
Scope:	Assembly unit
Syntax:	`-WmsgNe` <number>
Arguments:	<number>: Maximum number of error messages.
Default:	50
Description:	Sets the number of errors detected before the assembler stops processing
Example:	`ASMOPTIONS=-WmsgNe2`
	The assembler stops assembling after two error messages.
See also:	**5.4.17 -WmsgNi** and **5.4.18 -WmsgNw**

5.4.17 -WmsgNi

-WmsgNi:	Number of information messages
Group:	MESSAGES
Scope:	Assembly unit
Syntax:	`-WmsgNi`<number>
Arguments:	<number>: Maximum number of information messages.
Default:	50
Description:	Sets the maximum number of information messages to be logged
Example:	`ASMOPTIONS=-WmsgNi10`
	The first 10 information messages are logged.
See also:	**5.4.16 -WmsgNe** and **5.4.18 -WmsgNw**

5.4.18 -WmsgNw

-WmsgNw:	Number of warning messages
Group:	MESSAGES
Scope:	Assembly unit
Syntax:	`-WmsgNw` <number>
Arguments:	<number>: Maximum number of warning messages.
Default:	50
Description:	Sets the maximum number of warning messages to be logged
Example:	`ASMOPTIONS=-WmsgNw15`
	The first 15 warning messages are logged.
See also:	**5.4.16 -WmsgNe** and **5.4.17 -WmsgNi**

5.4.19 -WmsgFbv -WmsgFbm

-WmsgFb:	Set message file format for batch mode
Group:	MESSAGE
Scope:	Assembly unit
Syntax:	`-WmsgFb` [v | m]
Arguments:	v: Verbose format
	m: Microsoft format
Default:	-WmsgFbm

Description: The assembler can be started with additional arguments (for instance, files to be assembled together with assembler options). If the assembler has been started with arguments (for example, from the **Make** tool or with the `%f` argument from WinEdit), the assembler assembles the files in batch mode. No assembler window is visible and the assembler terminates after job completion.

If the assembler is in batch mode, assembler messages are written to a file instead of to the screen. This file contains only the assembler messages. By default, the assembler uses a Microsoft message format to write the assembler messages (errors, warnings, and information messages).

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

Example:

```
var1:     equ   5
var2:     equ   5
   if (var1=var2)
         nop
   endif
 endif
```

By default, the assembler generates this error information if it is running in batch mode:

```
X:\TW2.ASM(12):ERROR: conditional else not allowed here
```

Setting the format to verbose, more information is listed:

```
ASMOPTIONS=-WmsgFbv
>> in "X:\TW2.ASM", line 12, col 0, pos 215
      endif
      endif
     ^
ERROR A1001: Conditional else not allowed here
```

See also: **5.4.20 -WmsgFiv -WmsgFim**

5.4.20 -WmsgFiv -WmsgFim

-WmsgFi:	Set message file format for interactive mode
Group:	MESSAGE
Scope:	Assembly unit
Syntax:	`-WmsgFi` [v | m]
Arguments:	v: Verbose format m: Microsoft format
Default:	-WmsgFiv
Description:	If the assembler is started without additional arguments, it is in interactive mode (a window is visible). By default, the assembler uses the verbose error file format to write assembler messages (errors, warnings, and 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).

Example:

```
var1:     equ   5
var2:     equ   5
   if (var1=var2)
         nop
   endif
        endif
```

By default, the assembler generates the following error output in the assembler window if it is running in interactive mode.

```
>> in "X:\TWE.ASM", line 12, col 0, pos 215
         endif
         endif
^
ERROR A1001: Conditional else not allowed here
```

Setting the format to Microsoft, less information is displayed:

```
ASMOPTIONS=-WmsgFim
X:\TWE.ASM(12): ERROR: conditional else not allowed here
```

See also: **5.4.19 -WmsgFbv -WmsgFbm**

Section 6. Sections

6.1 Contents

6.2 Introduction 107

6.3 Section Attributes 108
6.3.1 Data Sections 108
6.3.2 Constant Data Sections 108
6.3.3 Code Sections 108

6.4 Section Types 109
6.4.1 Absolute Sections 109
6.4.2 Relocatable Sections 111
6.4.3 Relocatable versus Absolute Section 114
6.4.3.1 Modularity 114
6.4.3.2 Multiple Developers 114
6.4.3.3 Early Development 115
6.4.3.4 Enhanced Portability 115
6.4.3.5 Tracking Overlaps 115
6.4.3.6 Reusability 115

6.2 Introduction

Sections are portions of code or data that cannot be split into smaller elements. Each section has a name, type, and attributes. Each assembly source file contains at least one section.

The number of sections in an assembly source file is limited only by the amount of system memory available during assembly. If several sections with the same name are detected inside a single source file, the code is concatenated into one large section.

Sections with the same name, but from different modules, are combined into a single section when linked.

6.3 Section Attributes

According to content, an attribute is associated with each section. A section may be a:

- Data section
- Constant data section
- Code section

6.3.1 Data Sections

A section containing variables (variable defined using the DS (define space) directive) is considered to be a data section. Data sections are always allocated in the target processor RAM area.

Empty sections that do not contain any code or data declarations are also considered to be data sections.

6.3.2 Constant Data Sections

A section containing only constant data definitions (variables defined using the DC or DCB (define constance block) directives) is considered to be a constant section. Constant sections should be allocated in the target processor ROM area; otherwise, they cannot be initialized when the application is loaded.

NOTE: *It is strongly recommended that separate sections for definitions of variables and constant variables are defined. This will avoid any problems in the initialization of constant variables.*

6.3.3 Code Sections

A section containing at least an instruction is considered to be a code section. Code sections are always allocated in the target processor ROM area.

Code sections should not contain any variable definitions (variables defined using the DS directive). There is no write access on variables defined in a code section. Additionally, these variables cannot be displayed as data in the debugger.

6.4 Section Types

First, in an application, the programmer must decide which type of code to use:

- Absolute
- Relocatable

The assembler allows mixing of absolute and relocatable sections in a single application and also in a single source file. The main difference between absolute and relocatable sections is the way symbol addresses are determined.

6.4.1 Absolute Sections

The starting address of an absolute section is known at assembly time. An absolute section is defined by the directive ORG. The operand specified in the ORG directive determines the start address.

```
         XDEF entry
        ORG $040     ; Absolute constant data section.
cst1: DC.B    $26
cst2: DC.B    $BC
...
        ORG $080     ; Absolute data section.
var:  DS.B    1

        ORG $C00     ; Absolute code section.
entry:
        LDA  cst1    ; Load value in cst1
        ADD  cst2    ; Add value in cst2
        STA  var     ; Store in var
        BRA  entry
```

Figure 6-1. Absolute Section Programming Example

In the previous example, two bytes of storage are allocated starting at address `$040.` Symbol `cst1` will be allocated at address `$040` and `cst2` will be allocated at address `$041.` All subsequent instructions or data allocation directives will be located in the absolute section until another section is specified using the ORG or SECTION directive.

When using absolute sections, the user is responsible for ensuring that no overlap exists between the different absolute sections defined in the application. In the previous example, the programmer should ensure that the size of the section starting at address $040 is not bigger than $40 bytes; otherwise, the sections starting at $040 and $080 will overlap.

When object files are generated, applications containing only absolute sections must be linked. In that case, there should be no overlap between address ranges from the absolute sections defined in the assembly file and address ranges defined in the linker parameter file.

The PRM (parameter) file used to link the example in **Figure 6-1** is defined in **Figure 6-2**.

```
LINK test.abs  /* Name of the executable file generated. */
NAMES
  test.o       /* Name of object files in the application. */
END
SEGMENTS
/* READ_ONLY memory area. There should be no overlap between this
   memory area and the absolute sections defined in the assembly
   source file. */
  MY_ROM  = READ_ONLY  0x1000 TO 0x1FFF;
/* READ_WRITE memory area. There should be no overlap between this
   memory area and the absolute sections defined in the assembly
   source file. */
  MY_RAM  = READ_WRITE 0x2000 TO 0x2FFF;
END
PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM. */
  .data    INTO MY_RAM;
/*Relocatable code and constant sections are allocated in MY_ROM. */
  .text    INTO MY_ROM;
END
INIT entry                     /*  Application entry point */
VECTOR ADDRESS 0xFFFE entry /* initialization of the reset vector */
```

Figure 6-2. PRM File Example Code

The linker PRM file contains at least:

- The name of the absolute file (command LINK)
- The name of the object file that should be linked (command NAMES)
- Specification of a memory area where the sections containing variables must be allocated. At least the predefined section, `.data`, must be placed there (command SEGMENTS and PLACEMENT).
- Specification of a memory area where the sections containing code or constants must be allocated and at least the predefined section, `.text`, must be placed there (command SEGMENTS and PLACEMENT).
- The application entry point (command INIT)
- Definition of the reset vector (command VECTOR ADDRESS)

For applications containing only absolute sections, nothing will be allocated.

6.4.2 Relocatable Sections

The start address of a relocatable section is evaluated at link time, according to the information stored in the linker parameter file. A relocatable section is defined through the directive SECTION, as illustrated in **Figure 6-3**.

```
          XDEF entry
constSec: SECTION     ; Relocatable constant data
                      ;section
cst1:     DC.B $A6
cst2:     DC.B $BC
...
dataSec:  SECTION     ; Relocatable data section.
var:      DS.B 1

codeSec:  SECTION     ; Relocatable code section.
entry:
          LDA  cst1   ; Load value in cst1
          ADD  cst2   ; Add value in cst2
          STA  var    ; Store in var
          BRA  entry
```

Figure 6-3. Relocatable Section Programming Example

In the previous example, two bytes of storage are allocated in section `constSec`. Symbol `cst1` will be allocated at offset 0 and `cst2` at offset 1 from the beginning of the section. All subsequent instructions or data allocation directives will be located in the relocatable section `constSec` until another section is specified using the ORG or SECTION directive.

When using relocatable sections, the user does not need to worry about overlapping sections. The linker will assign a start address to each section according to the input from the linker parameter file.

The customer can define one memory area for the code and constant sections and another one for the variable sections or split sections over several memory areas.

When all constant and code sections as well as data sections can be allocated consecutively, the PRM file used to assemble the example in **Figure 6-3**, can be defined as follows:

```
LINK test.abs  /* Name of the executable file generated */
NAMES
  test.o       /* Name of object files in the application */
END
SEGMENTS
/* READ_ONLY memory area.  */
  MY_ROM  = READ_ONLY  0x0B00 TO 0x0BFF;
/* READ_WRITE memory area. */
  MY_RAM  = READ_WRITE 0x0080 TO 0x008F;
END
PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM. */
  .data    INTO MY_RAM;
/*Relocatable code and constant sections are allocated in MY_ROM */
  .text    INTO MY_ROM;
END
INIT entry               /*  Application entry point. */
VECTOR ADDRESS 0xFFFE entry /* initialization of the reset vector */
```

Figure 6-4. Defining One RAM and One ROM Area

According to the PRM file in **Figure 6-5**:

- The section `dataSec` will be allocated starting at `0x080`.
- The section `constSec` will be allocated starting at `0x0B00`.
- The section `codeSec` will be allocated next to the section `constSec`.

When the constant, code, and data sections cannot be allocated consecutively, the PRM file used to link the example above, can be defined as follows:

```
LINK test.abs  /* Name of the executable file generated. */
NAMES
  test.o       /* Name of the object files in the application. */
END
SEGMENTS
  ROM_AREA_1= READ_ONLY  0xB00 TO 0xB7F; /*READ_ONLY memory area. */
  ROM_AREA_2= READ_ONLY  0xC00 TO 0xC7F; /*READ_ONLY memory area. */
  RAM_AREA_1= READ_WRITE 0x800 TO 0x87F; /*READ_WRITE memory area.*/
  RAM_AREA_2= READ_WRITE 0x900 TO 0x97F; /*READ_WRITE memory area.*/
END
PLACEMENT
/*Relocatable variable sections are allocated in MY_RAM. */
  dataSec          INTO RAM_AREA_2;
  .data            INTO RAM_AREA_1;
/*Relocatable code and constant sections are allocated in MY_ROM. */
  constSec         INTO ROM_AREA_2;
  codeSec, .text   INTO ROM_AREA_1;
END
INIT entry              /*  Application entry point. */
VECTOR ADDRESS 0xFFFE entry /*initialization of the reset vector. */
```

Figure 6-5. Defining Multiple RAM and ROM Areas

According to the PRM file shown in **Figure 6-2**:

- The section `dataSec` will be allocated starting at `0x0900`.
- The section `constSec` will be allocated starting at `0x0C00`.
- The section `codeSec` will be allocated starting at `0x0B00`.

6.4.3 Relocatable versus Absolute Section

Generally, developing an application using relocatable sections is recommended. Relocatable sections offer several advantages.

6.4.3.1 Modularity

An application is more modular when programming can be divided into smaller units called sections. The sections themselves can be distributed among different source files.

6.4.3.2 Multiple Developers

When an application is split over different files, multiple developers can be involved in the development of the application. To avoid major problems when merging the different files, attention must be paid to the following items:

- An include file must be available for each assembly source file containing XREF directives for all exported variables, constants, and functions. Additionally, the interface to the function should be described at that point (parameter passing rules as well as function return value).
- When accessing variables, constants, or functions from another module, the corresponding include file must be included.
- Variables or constants defined by another developer must always be referenced by their names.
- Before invoking a function implemented in another file, the developer should respect the function interface; for instance, parameters are passed as expected and return value is retrieved correctly.

6.4.3.3 Early Development

The application can be developed before the application memory map is known. Often the definitive application memory map can be determined only after code and data size requirements can be evaluated. The size required for code or data can be quantified only after the major part of the application is implemented. When absolute sections are used, defining the definitive memory map is an iterative process of mapping and remapping the code. The assembly files must be edited, assembled, and linked several times. When relocatable sections are used, this can be achieved by editing the PRM file and linking the application.

6.4.3.4 Enhanced Portability

Because the memory map is not the same for all MCU derivatives, using relocatable sections allows the user to easily port the code for another MCU. When porting relocatable code to another target, link the application again with the appropriate memory map.

6.4.3.5 Tracking Overlaps

When using absolute sections, the programmer must ensure there is no overlap between sections. When using relocatable sections, the programmer does not need to be concerned about sections overlapping. The label offsets are evaluated relative to the beginning of the section. Absolute addresses are determined and assigned by the linker.

6.4.3.6 Reusability

When using relocatable sections, code implemented to handle a specific I/O (input/output) device (serial communication device) can be reused in another application without modification.

Section 7. Assembler Syntax

7.1 Contents

7.2 Introduction. 118

7.3 Comment Line 118

7.4 Source Line 119
7.4.1 Label Field 119
7.4.2 Operation Field 120
7.4.2.1 Instructions 120
7.4.2.2 Directives 125
7.4.2.3 Macro Names 125
7.4.3 Operand Fields 125
7.4.3.1 Inherent 126
7.4.3.2 Immediate 126
7.4.3.3 Direct. 127
7.4.3.4 Extended 128
7.4.3.5 Indexed, No Offset 129
7.4.3.6 Indexed, 8-Bit Offset. 130
7.4.3.7 Indexed, 16-Bit Offset. 131
7.4.3.8 Relative 132
7.4.4 Comment Field. 132

7.5 Symbols 132
7.5.1 User-Defined Symbols. 133
7.5.2 External Symbols. 134
7.5.3 Undefined Symbols 134
7.5.4 Reserved Symbols 135

7.6 Constants. 135
7.6.1 Integer Constants 135
7.6.2 String Constants. 136
7.6.3 Floating-Point Constants 136

7.7 Operators. 136
7.7.1 Addition and Subtraction Operators (Binary) 136
7.7.2 Multiplication, Division, and Modulo Operators (Binary) 137
7.7.3 Sign Operators (Unary) 137
7.7.4 Shift Operators (Binary). 138
7.7.5 Bitwise Operators (Binary) 138
7.7.6 Bitwise Operators (Unary). 139
7.7.7 Logical Operators (Unary). 139
7.7.8 Relational Operators (Binary) 140
7.7.9 HIGH Operator 141
7.7.10 LOW Operator 141
7.7.11 Force Operator (Unary) 142

7.8 Expressions 144
7.8.1 Absolute Expressions. 145
7.8.2 Simple Relocatable Expression 146

7.9 Translation Limits. 147

7.2 Introduction

An assembler source program is a sequence of source statements. Each source statement is coded on one line of text and can be a:

- Comment line
- Source line

7.3 Comment Line

A comment can occupy an entire line to explain the purpose and usage of a block of statements or to describe an algorithm. A comment line contains a semicolon followed by text. Comments are included in the assembly listing, but are not significant to the assembler.

An empty line is also considered to be a comment line.

Example:

```
; This is a comment line
```

7.4 Source Line

Each source statement includes one or more of these four fields:

1. A label
2. An operation field
3. One or several operands
4. A comment

Characters on the source line may be upper or lower case. Directives and instructions are case insensitive. Symbols are case sensitive, except when the CI option (specifying case insensitivity for label names) is activated.

7.4.1 Label Field

The label field is the first field in a source line. A label is a symbol followed by a colon. Labels can include letters, underscores, periods, and numbers. The first character must not be a number.

Labels are required on assembler directives that define the value of a symbol (SET or EQU). For these directives, labels are assigned the value corresponding to the expression in the operand field.

Labels specified in front of another directive, instruction, or comment are assigned the value of the location counter in the current section.

NOTE: *When the macro assembler expands a macro, it generates internal symbols starting with an underscore (`_`). Therefore, to avoid potential conflicts, user-defined symbols should not begin with an underscore.*

For the macro assembler, a `.B` at the end of a label means byte and a `.W` at the end of a label means word. Therefore, to avoid potential conflicts, user-defined symbols should not end with `.B` or `.W`.

7.4.2 Operation Field

The operation field follows the label field and is separated from it by a white space. The operation field must not begin in the first column.

An entry in the operation field is one of the following:

- An instruction mnemonic
- A directive name
- A macro name

7.4.2.1 Instructions

Executable instructions for the M68HC05 processor are defined in *CPU05 Reference Manual*, Motorola document order number CPU05RM/AD. **Table 7-1** presents a summary of available executable instructions.

Table 7-1. Executable Instructions (Sheet 1 of 5)

Instruction	Addressing Modes	Description
ADC	#<immediate> <direct or extended> <8 or 16 bit offset>,X ,X	Add with carry
ADD	#<immediate> <direct or extended> <8 or 16 bit offset>,X ,X	Add without carry
AND	#<immediate> <direct or extended> <8 or 16 bit offset>,X ,X	Logical AND
ASL	<direct> <8 bit offset>,X ,X	Arithmetic shift left
ASLA	No operands	Arithmetic shift left accumulator
ASLX	No operands	Arithmetic shift left register X
ASR	<direct> <8 bit offset>,X ,X	Arithmetic shift right
ASRA	No operands	Arithmetic shift right accumulator

Table 7-1. Executable Instructions (Sheet 2 of 5)

Instruction	Addressing Modes	Description
ASRX	No operands	Arithmetic shift right register X
BCC	<PC relative offset>	Branch if carry bit clear
BCLR	Bit Number, <direct>	Clear one bit in memory
BCS	<PC relative offset>	Branch if carry bit set
BEQ	<PC relative offset>	Branch if equal
BHCC	<PC relative offset>	Branch if half carry bit clear
BHCS	<PC relative offset>	Branch if half carry bit set
BHI	<PC relative offset>	Branch if higher
BHS	<PC relative offset>	Branch if higher or same
BIH	<PC relative offset>	Branch if /IRQ* pin high
BIL	<PC relative offset>	Branch if /IRQ* pin low
BIT	#<immediate> <direct or extended> <8 or 16 bit offset>,X ,X	Bit Test
BLO	<PC relative offset>	Branch if lower (same as BCS)
BLS	<PC relative offset>	Branch if lower or same
BMC	<PC relative offset>	Branch if interrupt mask clear
BMI	<PC relative offset>	Branch if minus
BMS	<PC relative offset>	Branch If interrupt mask set
BNE	<PC relative offset>	Branch if not equal
BPL	<PC relative offset>	Branch if plus
BRA	<PC relative offset>	Branch always
BRCLR	BitNumber,<direct>, <PC relative offset>	Branch if bit is clear
BRN	<offset>	Branch never
BRSET	BitNumber,<direct>, <PC relative offset>	Branch if bit set
BSET	BitNumber,<direct>	Set bit in memory
BSR	<PC relative offset>	Branch to subroutine
CLC	no operands	Clear carry bit
CLI	no operands	Clear interrupt mask bit

Table 7-1. Executable Instructions (Sheet 3 of 5)

Instruction	Addressing Modes	Description
CLR	<direct> <8 bit offset>,X ,X	Clear memory
CLRA	No operands	Clear accumulator A
CLRX	No operands	Clear index register X
CMP	#<immediate> <direct or extended> <8 or 16 bit offset>,X ,X	Compare accumulator with memory
CMPA	#<immediate> <direct or extended> <8 or 16 bit offset>,X ,X	Compare accumulator with memory (alias for CMP)
COM	<direct> <8 bit offset>,X ,X	One’s complement on memory location
COMA	No operands	One’s complement on accumulator A
COMX	No operands	One’s complement on register X
CPX	#<immediate> <direct or extended> <8 or 16 bit offset>,X ,X	Compare index register X with memory
DEC	<direct> <8 bit offset>,X ,X	Decrement memory location
DECA	No operands	Decrement accumulator
DECX	No operands	Decrement index register
EOR	#<immediate> <direct or extended> <8 or 16 bit offset>,X ,X	Exclusive OR memory with accumulator
INC	<direct> <8 bit offset>,X ,X	Increment memory location
INCA	No operands	Increment accumulator
INCX	No operands	Increment register X
JMP	<direct or extended> <8 or 16 bit offset>,X ,X	Jump to label

Table 7-1. Executable Instructions (Sheet 4 of 5)

Instruction	Addressing Modes	Description
JSR	<direct or extended> <8 or 16 bit offset>,X ,X	Jump to subroutine
LDA	#<immediate> <direct or extended> <8 or 16 bit offset>,X ,X	Load accumulator
LDX	#<immediate> <direct or extended> <8 or 16 bit offset>,X ,X	Load index register X from memory
LSL	<direct> <8 bit offset>,X ,X	Logical shift left in memory
LSLA	No operands	Logical shift left accumulator
LSLX	No operands	Logical shift left register X
LSR	<direct> <8 bit offset>,X ,X	Logical shift right in memory
LSRA	No operands	Logical shift right accumulator
LSRX	No operands	Logical shift right register X
MUL	No operands	Unsigned multiply
NEG	<direct> <8 bit offset>,X ,X	Performs a negate operation
NEGA	No operands	Two’s complement on accumulator
NEGX	No operands	Two’s complement on register X
NOP	No operands	No operation
ORA	#<immediate> <direct or extended> <8 or 16 bit offset>,X ,X	Inclusive OR between accumulator and memory
ROL	<direct> <8 bit offset>,X ,X	Rotate memory left
ROLA	No operands	Rotate accumulator left
ROLX	No operands	Rotate register X left

Table 7-1. Executable Instructions (Sheet 5 of 5)

Instruction	Addressing Modes	Description
ROR	<direct> <8 bit offset>,X ,X	Rotate memory right
RORA	No operands	Rotate accumulator right
RORX	No operands	Rotate register X right
RSP	No operands	Reset stack pointer
RTI	No operands	Return from Interrupt
RTS	No operands	Return from subroutine
SBC	#<immediate> <direct or extended> <8 or 16 bit offset>,X ,X	Subtract with carry
SEC	No operands	Set carry bit
SEI	No operands	Set interrupt mask bit
STA	<direct or extended> <8 or 16 bit offset>,X ,X	Store accumulator in memory
STOP	No operands	Enable /IRQ* pin and stop oscillator
STX	<direct or extended> <8 or 16 bit offset>,X ,X	Store index register X in memory
SUB	#<immediate> <direct or extended> <8 or 16 bit offset>,X ,X	Subtract
SWI	No operands	Software interrupt
TAX	No operands	Transfer accumulator to index register X
TST	<direct> <8 bit offset>,X ,X	Test memory for negative or 0
TSTA	No operands	Test accumulator for negative or 0
TSTX	No operands	Test register X for negative or 0
TXA	No operands	Transfer index register X to accumulator
WAIT	No operands	Enable interrupts; stop processor

7.4.2.2 Directives

Assembler directives are described in **Section 8. Assembler Directives**.

7.4.2.3 Macro Names

A user-defined macro can be invoked in the assembler source program. This results in the expansion of the code defined in the macro. Defining and using macros are described in **Section 9. Macros**.

7.4.3 Operand Fields

The operand fields, when present, follow the operation field and are separated from it by a white space. When two or more operand subfields appear within a statement, a comma must separate them. The addressing mode notations shown in **Table 7-2** are allowed in the operand field.

Table 7-2. Addressing Mode Notations

Addressing Mode	Notation	Example
Inherent	No operands	RSP
Direct	<8-bit address>	ADC byte
Extended	<16-bit address>	ADC word
Relative	<PC relative offset>	BRA Label
Immediate	#<values>	ADC #$01
Indexed, no offset	,X	ADC ,X
Indexed, 8-bit offset	<8-bit offset>,X	ADC Offset,X
Indexed, 16-bit offset	<16-bit offset>,X	ADC Offset,X

7.4.3.1 Inherent

Instructions using this addressing mode either have no operands or all operands are stored in internal CPU registers. The CPU does not need to perform memory access to complete the instruction.

Example:

```
CLRA
CLRX
```

7.4.3.2 Immediate

The opcode contains the value to use with the instruction rather than the address of this value. The # (pound) character indicates an immediate addressing operand.

Example:

```
           XDEF Entry

MyData:    SECTION
data:      DS.B 1

MyCode:    SECTION
Entry:
            RSP           ; init Stack Pointer

main:      LDA #$50
           BRA main
```

In this example, the hexadecimal value `$50` is loaded in register `A`. The code for this instruction is `A650` where A6 is the opcode of the `LDA` instruction and `$50` the immediate value to load in register `A`.

The immediate addressing mode can also be used to refer to the address of a symbol.

Example:

```
           ORG $80
var1:      DC.B $45, $67
           ORG $800
main:
           LDX  #var1
           BRA  main
```

In this example, the address of the variable `var1 ($80)` is loaded in register X. One very common programming error is to omit the # character. This causes the assembler to misinterpret the expression as an address rather than explicit data.

Example:

```
LDA   $60
```

means load accumulator A with the value stored at address `$60`.

7.4.3.3 Direct

The direct addressing mode is used to address operands in the direct page of the memory (location $0000 to $00FF). Access to this memory range (also called zero page) is faster and requires less code than the extended addressing mode (see the following code example). To speed up an application, a programmer can place the most commonly accessed data in this area of memory. For most of the direct instructions, only two bytes are required. The first byte is the opcode and the second byte is the operand address located in page zero.

Example:

```
           XDEF Entry

MyData:    SECTION SHORT
data:      DS.B 1

MyCode:    SECTION
Entry:
           RSP              ; init Stack Pointer
           LDA #$01
main:      STA data
           INCA
           BRA main
```

In this example, the value in register A is stored in the variable data located on the direct page. The value in A is then incremented. The section MyData must be defined in the direct page in the linker parameter file. The opcode generated for the instruction `STA data` is two bytes long.

7.4.3.4 Extended

The extended addressing mode is used to access memory located above the direct page in a 64-Kbyte memory map. For the extended instructions, three bytes are required. The first byte is the opcode and the second and third bytes are the most and least significant bytes of the operand address.

Example:

```
           XDEF Entry
           ORG $100
data:      DS.B 1
MyCode:    SECTION
Entry:
           RSP              ; init Stack Pointer
           LDA #$01
main:      STA data
           INCA
           BRA main
```

In this example, the value in register A is stored in the variable data. This variable is located at address $0100 in the memory map. The value in A is then incremented. The opcode of the instruction STA data is now three bytes long.

7.4.3.5 Indexed, No Offset

This addressing mode is used to access data with variable addresses through the index X register of the controller. The index X register contains the address of the direct data. All indexed, no offset instructions are one byte long.

Example:

```
            XDEF Entry
RST_VECTOR: EQU  $FFFE
PAGE0_RAM:  EQU  $80
PAGE0_ROM:  EQU  $40
            ORG RST_VECTOR    ; Reset Vector
Reset:      DC.W Entry        ; initialization
            ORG PAGE0_RAM
data:       DS.B 4
            ORG PAGE0_ROM
Entry:      RSP               ; Stack pointer
            LDA #$55          ; initialization

            STA data+0
            STA data+1
            STA data+2
            STA data+3
main:
            LDX #data+2
            INC ,X

            LDX #main
            JMP ,X
```

The value stored in memory location `data+2` is incremented. The JMP instruction causes the program to jump to the address pointed to by the X register: main.

7.4.3.6 *Indexed, 8-Bit Offset*

This addressing mode is useful when selecting the kth element in an n-element table. The size of the table is limited to the first 511 bytes of memory. The sum of the unsigned byte in the index register added to the unsigned byte following the opcode is the conditional address of the operand.

Example:

```
                          XDEF Entry
     RST_VECTOR: EQU  $FFFE
     PAGE0_ROM:  EQU  $40
     TABLE_LOC:  EQU  $E0

                 ORG RST_VECTOR    ; Reset Vector
     Reset:      DC.W Entry        ; initialization

                 ORG TABLE_LOC
     TABLE:      DS.B 5

                 ORG PAGE0_ROM
     Entry:      RSP               ; Stack pointer
                                   ; initialization
                 LDA #$00
                 STA TABLE+0       ; initialize TABLE1
                 STA TABLE+1
                 STA TABLE+2
                 STA TABLE+3
                 STA TABLE+4

     main:
                 LDX #2            ; value in A is
                 STA TABLE_LOC,X   ; stored in TABLE1[2]

                 INCA
                 BRA main
```

`TABLE` is defined at address `$E0` in the direct memory area. All elements of this table are first initialized to 0 using the direct addressing mode. The value in the third element of the table (`TABLE[2]`) is then incremented. Register X contains the offset of the data to access in the `TABLE`. The offset is the address of the first element of the table in memory. Since the maximum value for the offset is $FF, the maximum number of elements should be $1FE. Do not overlap the stack defined at address $FF.

7.4.3.7 Indexed, 16-Bit Offset

This addressing mode is useful when selecting the kth element in an n-element table. The size of the table is limited to `$FFFF` bytes. Indexed, 16-bit offset instructions are three bytes long. The sum of the unsigned byte in the index register added to the two unsigned bytes following the opcode is the conditional address of the operand.

Example:

```
            XDEF Entry
RST_VECTOR: EQU  $FFFE
ROM:        EQU  $B00
TABLE_LOC:  EQU  $C00
            ORG RST_VECTOR    ; Reset Vector
Reset:      DC.W Entry        ; initialization
            ORG TABLE_LOC
TABLE:      DS.B 5

            ORG ROM
Entry:      RSP               ; Stack pointer
                              ; initialization
            LDA #$00
            STA TABLE+0       ; initialize TABLE
            STA TABLE+1
            STA TABLE+2
            STA TABLE+3
            STA TABLE+4

main:
            LDX #4            ; value in A is
            STA TABLE_LOC,X   ; stored in TABLE[4]

            INCA
            BRA main
```

`TABLE` is defined at address $C00 in the direct memory area. All elements of this table are first initialized to 0 using the direct addressing mode. The value in the third element of the table (`TABLE[4]`, memory location `$C04`) is then incremented. Register X contains the offset of the data to access in the `TABLE`. The offset is the address of the first element of the table in memory.

7.4.3.8 Relative

This addressing mode is used by all branch instructions to determine the destination address. The signed byte following the opcode is added to the contents of the program counter to form the effective branch address.

Since the offset is coded on a signed byte, the branching offset range is –127 to +128. The destination address of the branch instruction must be in this range.

Example:

```
main:
      NOP
      NOP
      BRA main
```

In this example, after the two NOPs have been executed, the application branches at offset –2 from the BRA instruction (on label `main`).

7.4.4 Comment Field

The last field in a source statement is an optional comment field. A semicolon (;) is the first character in a comment field.

Example:

```
NOP  ; Comment following an instruction
```

7.5 Symbols

The following sections describe symbols used by the assembler.

7.5.1 User-Defined Symbols

Symbols identify memory locations in program or data sections in an assembly module. A symbol has two attributes:

1. The section in which the memory location is defined
2. The offset from the beginning of that section

Symbols can be defined with an absolute or relocatable value, depending on the section in which the labeled memory location is found. If the memory location is located within a relocatable section (defined with the SECTION directive), the label has a relocatable value relative to the section start address.

Symbols can be defined relocatable in the label field of an instruction or data definition source line. In the next example, `labelx` is used to represent a symbol.

```
Sec: SECTION
label1: DC.B 2 ; label1 is assigned offset 0 within Sec
label2: DC.B 5 ; label2 is assigned offset 2 within Sec
label3: DC.B 1 ; label3 is assigned offset 7 within Sec
```

Figure 7-1. Relocatable Symbols Program Example

It is also possible to define a label with either an absolute or a previously defined relocatable value, using a SET or EQU directive.

Symbols with absolute values must be defined with constant expressions.

```
Sec: SECTION
label1: DC.B 2     ; label1 is assigned offset 0 within ;;
                   Section
label2: EQU 5      ; label2 is assigned value 5
label3: EQU label1 ; label3 is assigned address of label1
```

Figure 7-2. SET or EQU Directive Program Example

7.5.2 External Symbols

A symbol can be made external using the XDEF directive. In another source file, an XREF or XREFB directive may reference it. Since its address is unknown in the referencing file, it is considered to be relocatable.

```
XREF extLabel  ; symbol defined in an other module.
               ; extLabel is imported in current module
XDEF label     ; symbol made external for other modules
               ;label is exported from current module
constSec: SECTION
label:    DC.W 1, extLabel
```

Figure 7-3. External Symbol Program Example

7.5.3 Undefined Symbols

If a label is neither defined in the source file nor declared external using XREF or XREFB, the assembler considers it to be undefined and generates an error.

```
codeSec: SECTION
entry:
     NOP
     BNE  entry
     NOP
     JMP  end
     JMP  label <- Undeclared user-defined symbol : label
end: RTS
     END
```

Figure 7-4. Undefined Symbol Example

7.5.4 Reserved Symbols

Reserved symbols cannot be used for use-defined symbols. Register names are reserved identifiers. Reserved identifiers for the HC05 microcontroller are:

```
A, CCR, X, SP
```

Keywords LOW and HIGH are also reserved identifiers. They refer to the low and high byte of any specified memory location.

7.6 Constants

The assembler supports integer and ASCII string constants.

7.6.1 Integer Constants

The assembler supports four representations of integer constants:

1. A decimal constant is defined by a sequence of decimal digits (0–9).
   Example: 5, 512, 1024
2. A hexadecimal constant is defined by a dollar character ($) followed by a sequence of hexadecimal digits (0–9, a–f, A–F).
   Example: $5, $200, $400
3. An octal constant is defined by the at character (@) followed by a sequence of octal digits (0–7).
   Example: @5, @1000, @2000
4. A binary constant is defined by a percent (%) character followed by a sequence of binary digits (0–1).
   Example: %101, %1000000000, %10000000000

The default base for integer constants is initially decimal, but it can be changed using the BASE directive. When the default base is not decimal, decimal values cannot be represented because they do not have a prefix character.

7.6.2 String Constants

A string constant is a series of printable characters enclosed in single (‘) or double quotes (“). Double quotes are allowed only within strings delimited by single quotes. Single quotes are allowed only within strings delimited by double quotes.

Example**:**

```
'ABCD', "ABCD", 'A', "'B", "A'B", 'A"B'
```

7.6.3 Floating-Point Constants

The macro assembler does not support floating-point constants.

7.7 Operators

The following subsections describe the operators used in expressions the assembler recognizes.

7.7.1 Addition and Subtraction Operators (Binary)

Syntax: Addition: <*operand*> + <*operand*>
Subtraction: <*operand*> – <*operand*>

Description: The + (plus) operator adds two operands, whereas the – (minus) operator subtracts them. The operands can be any expression evaluating to an absolute or relocatable expression.

NOTE: *Addition between two relocatable operands is not allowed.*

Example:

```
$A3216 + $42; Addition of two absolute operands (= $A3258)
label - $10 ; Subtraction with value of label
```

7.7.2 Multiplication, Division, and Modulo Operators (Binary)

Syntax: Multiplication: <operand> * <operand>
Division: <operand> / <operand>
Modulo: <operand> % <operand>

Description: The * (asterisk) operator multiplies two operands, and the / (slash) operator performs an integer division of the two operands and returns the quotient of the operation. The % (percent) operator performs an integer division of the two operands and returns the remainder of the operation.

The operands can be any expression evaluating to an absolute expression. The second operand in a division or modulo operation cannot be 0.

Example:

```
23 * 4    ; multiplication ( = 92)
23 / 4    ; division ( = 5)
23 % 4    ; remainder( = 3)
```

7.7.3 Sign Operators (Unary)

Syntax: Plus: +<operand>
Minus: –<operand>

Description: The + (plus) operator does not change the operand, whereas the – (minus) operator changes the operand to its two’s complement. These operators are only valid for absolute expression operands.

Example:

```
+$32       ; ( = $32)
-$32       ; ( = $CE = -$32)
```

7.7.4 Shift Operators (Binary)

Syntax: Shift left: <operand> << <count>
Shift right: <operand> >> <count>

Description: The << (double less than) operator shifts left operand left by the number of bytes specified in the right operand.

The >> (double greater than) operator shifts left operand right by the number of bytes specified in the right operand.

The operands can be any expression evaluating to an absolute expression.

Example:

```
$25 << 2  ; shift left ( = $94)
$A5 >> 3  ; shift right( = $14)
```

7.7.5 Bitwise Operators (Binary)

Syntax: Bitwise AND: <operand> & <operand>
Bitwise OR: <operand> | <operand>
Bitwise XOR: <operand> ^ <operand>

Description: The & (ampersand) operator performs an AND between the two operands at the bit level.

The | (vertical bar) operator performs an OR between the two operands at the bit level.

The ^ (caret) operator performs an XOR between the two operands at the bit level.

The operands can be any expression evaluating to an absolute expression.

Example:

```
$E & 3    ; = $2 (%1110 & %0011 = %0010)
$E | 3    ; = $F (%1110 | %0011 = %1111)
$E ^ 3    ; = $D (%1110 ^ %0011 = %1101)
```

7.7.6 Bitwise Operators (Unary)

Syntax:	One’s complement: ~<operand>
Description:	The ~ (tilde) operator evaluates the one’s complement of the operand.
	The operand can be any expression evaluating to an absolute expression.

Example:

```
~$C   ; = $FFFFFFF3 (~%00000000 00000000 00000000 00001100
                     =%11111111 11111111 11111111 11110011)
```

7.7.7 Logical Operators (Unary)

Syntax:	Logical NOT: !<operand>
Description:	The ! (exclamation point) operator returns 1 (true) if the operand is 0; otherwise, it returns 0 (false).
	The operand can be any expression evaluating to an absolute expression.

Example:

```
!(8<5)     ; = $1 (TRUE)
```

7.7.8 Relational Operators (Binary)

Syntax:

Equal:	<operand> = <operand>
	<operand> == <operand>
Not equal:	<operand> != <operand>
	<operand> <> <operand>
Less than:	<operand> < <operand>
Less than or equal:	<operand> <= <operand>
Greater than:	<operand> > <operand>
Greater than or equal:	<operand> >= <operand>

Description: These operators compare the two operands and return 1 if the condition is true or 0 if the condition is false.

The operands can be any expression evaluating to an absolute expression.

Example:

```
3 >= 4       ; = 0  (FALSE)
label = 4    ; = 1  (TRUE) if label is 4, 0
             ; (FALSE) otherwise.
9 <  $B      ; = 1  (TRUE)
```

7.7.9 HIGH Operator

Syntax:	High Byte: HIGH(<operand>)
Description:	This operator returns the high byte of a memory address.
Example:	Assume `data1` is a word located at address `$1050` in memory.

```
LDA   #HIGH(data1)
```

This instruction loads the high byte of the address `data1` (`$10`) in register `A`.

```
LDA   HIGH(data1)
```

This instruction will load the direct value at the memory location of the higher byte of the address `data1` (for instance, the value in memory location `$10`) into register `A`.

7.7.10 LOW Operator

Syntax:	LOW Byte: LOW(<operand>)
Description	This operator returns the low byte of a memory address.
Example:	Assume `data1` is a word located at address `$1050` in memory.

```
LDA   #LOW(data1)
```

This instruction will load the immediate value of the lower byte of the address `data1` (`$50`) into register `A`.

```
LDA   LOW(data1)
```

This instruction will load the direct value of the lower byte of the address of `data1` (value in memory location `$50`) into register `A`.

7.7.11 Force Operator (Unary)

Syntax:	8-bit address:	<<operand>
		<operand>.B
	16-bit address: >	<operand>
		<operand>.W

Description: The `<` or `.B` operators force the operand to be an 8-bit operand, whereas the `>` or `.W` operators force the operand to be a 16-bit operand.

The `<` operator may be useful to force the 8-bit immediate, indexed, or direct addressing mode for an instruction.

The `>` operator may be useful to force the 16-bit immediate, indexed, or extended addressing mode for an instruction.

The operand can be any expression evaluating to an absolute or relocatable expression.

Example:

```
<label   ; label is an 8-bit address.
label.B  ; label is an 8-bit address.
>label   ; label is a 16-bit address.
label.W  ; label is a 16-bit address.
```

Operator precedence follows the rules for ANSI C operators.

Table 7-3. Operator Precedence

Operator	Description	Associativity
()	Parenthesis	Right to left
< > .B .W PAGE	Force direct address, index or immediate value to 8 bits Force direct address, index or immediate value to 16 bits Force direct addressing mode for absolute address. Force extended addressing mode for absolute address Access 4-bit page number (bits 16–19 of 20-bit value).	Right to left
~ + –	One's complement Unary plus Unary minus	Left to right
* / %	Integer multiplication Integer division Integer modulo	Left to right
+ –	Integer addition Integer subtraction	Left to right
<< >>	Shift left Shift right	Left to right
< <= > >=	Less than Less or equal to Greater than Greater or equal to	Left to right
=, == !=, <>	Equal to Not equal to	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise OR	Left to right

7.8 Expressions

An expression is composed of one or more symbols or constants, which are combined with unary or binary operators. Valid symbols in expressions are:

- User-defined symbols
- External symbols
- The special symbol * (asterisk) represents the value of the location counter at the beginning of the instruction or directive, even when several arguments are specified. In this example, the asterisk represents the location counter at the beginning of the `DC` directive:

```
    DC.W   1, 2, *-2
```

Once a valid expression has been fully evaluated by the assembler, it is reduced to one of three types of expressions:

1. Absolute expression — The expression has been reduced to an absolute value, which is independent of the start address of any relocatable section. Thus, it is a constant.
2. Simple relocatable expression — The expression evaluates to an absolute offset from the start of a single relocatable section.
3. Complex relocatable expression — The expression neither evaluates to an absolute expression nor to a simple relocatable expression. The assembler does not support such expressions.

All valid user-defined symbols representing memory locations are simple relocatable expressions. This includes labels specified in XREF directives, which are assumed to be relocatable symbols.

7.8.1 Absolute Expressions

Expressions involving constants, known absolute labels, or expressions are absolute expressions. An expression containing an operation between an absolute expression and a constant value is also an absolute expression.

Example of absolute expression:

```
Base:  SET $100
Label: EQU Base * $5 + 3
```

Expressions involving the difference between two relocatable symbols defined in the same file and in the same section evaluate to an absolute expression. An expression such as `label2-label1` can be translated as:

```
(<offset label2> + <start section address >) –
(<offset label1> + <start section address >)
```

This can be simplified as:

```
<offset label2> + <start section address > –
<offset label1> - <start section address>
= <offset label2> - <offset label1>
```

In the next example, the expression `tabEnd-tabBegin` evaluates to an absolute expression and is assigned the value of the difference between the offset of `tabEnd` and `tabBegin` in the section `DataSec`.

```
DataSec:   SECTION
tabBegin:  DS.B  5
tabEnd:    DS.B  1

CodeSec:   SECTION
entry:     LDD  #tabEnd-tabBegin  <- Absolute expression
```

7.8.2 Simple Relocatable Expression

A simple relocatable expression results from operations such as those shown here:

- <relocatable expression> + <absolute expression>
- <relocatable expression> – <absolute expression>
- < absolute expression> + < relocatable expression>

Example:

```
           XREF XtrnLabel
DataSec:   SECTION
tabBegin:  DS.B  5
tabEnd:    DS.B  1

CodeSec:   SECTION
entry:     LDA tabBegin+2  <- Simple relocatable expression
           BRA *-3         <- Simple relocatable expression
           LDA XtrnLabel+6 <- Simple relocatable expression
```

Table 7-4 describes the type of expression according to the operator in an unary operation:

Table 7-4. Expression — Operator Relationship (Unary)

Operator	Operand	Expression
–, !, ~	Absolute	Absolute
–, !, ~	Relocatable	Complex
+	Absolute	Absolute
+	Relocatable	Relocatable

Table 7-5 describes the type of expression according to left and right operators in a binary operation:

Table 7-5. Expression — Operator Relationship (Binary Operation)

Operator	Left Operand	Right Operand	Expression
–	Absolute	Absolute	Absolute
–	Relocatable	Absolute	Relocatable
–	Absolute	Relocatable	Complex
–	Relocatable	Relocatable	Absolute
+	Absolute	Absolute	Absolute
+	Relocatable	Absolute	Relocatable
+	Absolute	Relocatable	Relocatable
+	Relocatable	Relocatable	Complex
*, /, %, <<, >>, |, &, ^	Absolute	Absolute	Absolute
*, /, %, <<, >>, |, &, ^	Relocatable	Absolute	Complex
*, /, %, <<, >>, |, &, ^	Absolute	Relocatable	Complex
*, /, %, <<, >>, |, &, ^	Relocatable	Relocatable	Complex

7.9 Translation Limits

These limitations apply to the macro assembler:

- Floating-point constants are not supported.
- Complex relocatable expressions are not supported.
- Lists of operands or symbols must be separated with a comma.
- Include may be nested up to 50.
- The maximum line length is 1023.

Section 8. Assembler Directives

8.1 Contents

8.2 Introduction. 151

8.3 Directive Overview. 151
8.3.1 Section Definition Directives. 151
8.3.2 Constant Definition Directives. 151
8.3.3 Data Allocation Directives. 152
8.3.4 Symbol Linkage Directives 152
8.3.5 Assembly Control Directives. 152
8.3.6 Listing File Control Directives 153
8.3.7 Macro Control Directives. 154
8.3.8 Conditional Assembly Directives 154

8.4 ABSENTRY — Application Entry Point 155

8.5 ALIGN — Align Location Counter 156

8.6 BASE — Set Number Base 157

8.7 CLIST — List Conditional Assembly 158

8.8 DC — Define Constant. 160

8.9 DCB — Define Constant Block 162

8.10 DS — Define Space 163

8.11 ELSE — Conditional Assembly. 164

8.12 END — End Assembly. 165

8.13 ENDIF — End Conditional Assembly 166

8.14 ENDM — End Macro Definition 166

8.15 EQU — Equate Symbol Value 167

8.16 EVEN — Force Word Alignment. 168

8.17 FAIL — Generate Error Message 169
8.18 IF — Conditional Assembly 172
8.19 IFCC — Conditional Assembly 173
8.20 INCLUDE — Include Text from Another File. 175
8.21 LIST — Enable Listing. 175
8.22 LLEN — Set Line Length. 177
8.23 LONGEVEN — Forcing Longword Alignment. 178
8.24 MACRO — Begin Macro Definition 179
8.25 MEXIT — Terminate Macro Expansion 180
8.26 MLIST — List Macro Expansions 182
8.27 NOLIST — Disable Listing 185
8.28 NOPAGE — Disable Paging 186
8.29 ORG — Set Location Counter 186
8.30 OFFSET — Create Absolute Symbols 186
8.31 PAGE — Insert Page Break 188
8.32 PLEN — Set Page Length 189
8.33 SECTION — Declare Relocatable Section 189
8.34 SET — Set Symbol Value 191
8.35 SPC — Insert Blank Lines 192
8.36 TABS — Set Tab Length 192
8.37 TITLE — Provide Listing Title 192
8.38 XDEF — External Symbol Definition 193
8.39 XREF — External Symbol Reference 194

8.2 Introduction

This section introduces assembler directives. Functional descriptions and examples of each directive are provided.

8.3 Directive Overview

There are different classes of assembler directives. **Table 8-1** and **Table 8-2** give an overview of the different directives.

8.3.1 Section Definition Directives

The directives in **Table 8-1** define new sections.

Table 8-1. Section Directives

Directive	Description
ORG	Defines an absolute section
SECTION	Defines a relocatable section
OFFSET	Defines an offset section

8.3.2 Constant Definition Directives

The directives in **Table 8-2** define assembly constants.

Table 8-2. Constant Directives

Directive	Description
EQU	Assigns a name to an expression (cannot be redefined)
SET	Assigns a name to an expression (can be redefined)

8.3.3 Data Allocation Directives

The directives in **Table 8-3** allocate variables.

Table 8-3. Data Allocation Directives

Directive	Description
DC	Defines a constant variable
DCB	Defines a constant block
DS	Defines storage for a variable

8.3.4 Symbol Linkage Directives

The directives in **Table 8-4** export or import global symbols.

Table 8-4. Symbol Linkage Directives

Directive	Description
ABSENTRY	Specify the application entry point when an absolute file is generated
XDEF	Make a symbol public (visible from outside)
XREF	Import reference to an external symbol
XREFB	Import reference to an external symbol located on the direct page

8.3.5 Assembly Control Directives

Table 8-5 lists general-purpose directives that are used to control the assembly process.

Table 8-5. Assembly Control Directives

Directive	Description
ALIGN	Define alignment constraint
BASE	Specify default base for constant definition
END	End of assembly unit
EVEN	Define 2-byte alignment constraint

Table 8-5. Assembly Control Directives (Continued)

Directive	Description
FAIL	Generate user-defined error or warning messages
INCLUDE	Include text from another file
LONGEVEN	Define 4-byte alignment constraint

8.3.6 Listing File Control Directives

The directives in **Table 8-6** control the generation of the assembler listing file.

Table 8-6. Assembler List File Directives

Directive	Description
CLIST	Specify if all instructions in a conditional assembly block must be inserted in the listing file or not
LIST	Specify that all subsequent instructions must be inserted in the listing file
LLEN	Define line length in assembly listing file
MLIST	Specify if macro expansions must be inserted in the listing file
NOLIST	Specify that all subsequent instructions must not be inserted in the listing file
NOPAGE	Disable paging in the assembly listing file
PAGE	Insert page break
PLEN	Define page length in the assembler listing file
SPC	Insert an empty line in the assembly listing file
TABS	Define number of characters to insert in the assembler listing file for a TAB character
TITLE	Define the user-defined title for the assembler listing file

8.3.7 Macro Control Directives

The directives in **Table 8-7** are used for the definition and expansion of macros.

Table 8-7. Macro Directives

Directive	Description
ENDM	End of user-defined macro
MACRO	Start of user-defined macro
MEXIT	Exit from macro expansion

8.3.8 Conditional Assembly Directives

The directives in **Table 8-8** are used for conditional assembly.

Table 8-8. Conditional Assembly Directives

Directive	Description
ELSE	Alternate of conditional block
ENDIF	End of conditional block
IF	Start of conditional block. A Boolean expression follows this directive.
IFC	Test if two string expressions are equal
IFDEF	Test if a symbol is defined
IFEQ	Test if an expression is null
IFGE	Test if an expression is greater than or equal to 0
IFGT	Test if an expression is greater than 0
IFLE	Test if an expression is less than or equal to 0.
IFLT	Test if an expression is less than 0
IFNC	Test if two string expressions are different
IFNDEF	Test if a symbol is undefined
IFNE	Test if an expression is not null

8.4 ABSENTRY — Application Entry Point

Syntax: `ABSENTRY` <label>

Description: This directive specifies the application entry point in a directly generated absolute file (the option -FA2 `ELF/DWARF 2.0` absolute file must be enabled).

Using this directive, the entry point of the assembly application is written in the ELF header of the generated absolute file. When this file is loaded in the debugger, the line where the entry point label is defined is highlighted in the source window.

Example: If the next example is assembled using the -FA2 option, an `ELF/DWARF 2.0` absolute file is generated.

```
        ABSENTRY entry

        ORG $fffe
Reset:  DC.W entry

        ORG $70
entry:  NOP
        NOP
main:   RSP
        NOP
        BRA main
```

8.5 ALIGN — Align Location Counter

Syntax: `ALIGN` <n>

Description: This directive forces the next instruction to a boundary that is a multiple of <n>, relative to the start of the section. The value of <n> must be a positive number between 1 and 32,767. The `ALIGN` directive can force alignment to any size. The filling bytes inserted for alignment purpose are initialized with \0.

`ALIGN` can be used in code or data sections.

Example: The following example aligns the `HEX` label to a location, which is a multiple of 16 (in this case, location 00010 (hex))

```
000000 4849 4748   DC.B  "HIGH"
000004 0000 0000   ALIGN 16
000008 0000 0000
00000C 0000 0000
000010 007F          HEX:  DC.W  127  ; HEX is allocated on an
                                      ; address, a multiple of 16.
```

8.6 BASE — Set Number Base

Syntax: BASE <n>

Description: This directive sets the default number base for constants to <n>. The operand <n> may be prefixed to indicate its number base; otherwise, the operand is considered to be in the current default base. Valid values of <n> are 2, 8, 10, 16. Unless a default base is specified using the BASE directive, the default number base is decimal.

Example:

```
4    4                      base    10   ; default base is decimal
5    5   000000 64          dc.b    100
6    6                      base    16   ; default base is hex
7    7   000001 0A          dc.b    0a
8    8                      base    2    ; default base is binary
9    9   000002 04          dc.b    100
10   10  000003 04          dc.b    %100
11   11                     base    @12  ; default base is decimal
12   12  000004 64          dc.b    100
13   13                     base    $a   ; default base is decimal
14   14  000005 64          dc.b    100
15   15
16   16                     base    8    ; default base is octal
17   17  000006 40          dc.b    100
```

NOTE: *If the base value is set to 16, hexadecimal constants terminated by a* `D` *must be prefixed by the* `$` *(dollar sign) character; otherwise, they are interpreted as decimal constants in old style format. For example, constant 45D is interpreted as decimal constant 45, not as the hexadecimal constant $45D.*

8.7 CLIST — List Conditional Assembly

Syntax: `CLIST` [ON | OFF]

Description: The `CLIST` directive controls the listing of subsequent conditional assembly blocks. It precedes the first directive of the conditional assembly block to which it applies and remains effective until the next `CLIST` directive is read.

When `ON` is specified in a `CLIST` directive, the listing file includes all directives and instructions in the conditional assembly block, even those that are skipped and do not generate code.

When `OFF` is specified, only the directives and instructions that generate code are listed. When the option `–L` is activated, the assembler defaults to `CLIST ON`.

Example: Listing file with `CLIST OFF:`

```
         CLIST OFF
Try:      EQU   0
         IFEQ  Try
           LDA #103
         ELSE
           LDA #0
         ENDIF
```

The corresponding listing file is:

```
Motorola HC05-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997

Abs. Rel.   Loc.  Obj. code   Source line
---- ----   ------ ---------   -----------
 2    2          0000 0000    Try:    EQU   0
 3    3          0000 0000            IFEQ  Try
 4    4      000000 A667                  LDA #103
 5    5                              ELSE
 7    7                              ENDIF
 8    8
```

Listing file with `CLIST ON`:

When assembling the code:

```
         CLIST ON
Try:      EQU   0
         IFEQ  Try
           LDA #103
         ELSE
           LDA #0
         ENDIF
```

The corresponding listing file is:

```
Motorola HC05-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997

Abs. Rel.   Loc.  Obj. code   Source line
---- ----   ------ ---------   -----------
 2    2          0000 0000   Try:     EQU   0
 3    3          0000 0000            IFEQ  Try
 4    4   000000 A667                  LDA #103
 5    5                              ELSE
 6    6                                LDA #0
 7    7                              ENDIF
 8    8
```

8.8 DC — Define Constant

Syntax: [<label>:] DC [<size>] <expression> [, <expression>]...

where
<size> = B (default), W, or L

Description: The DC directive defines constants in memory. It can have one or more <expression> operands, which are separated by commas. The <expression> can contain an actual value (binary, octal, decimal, hexadecimal, or ASCII). Alternately, the <expression> can be a symbol or expression that can be evaluated by the assembler as an absolute or simple relocatable expression. One memory block is allocated and initialized for each expression.

These rules apply to size specifications for DC directives:

- DC.B — One byte is allocated for numeric expressions. One byte is allocated per ASCII character for strings.
- DC.W — Two bytes are allocated for numeric expressions. ASCII strings are right aligned on a 2-byte boundary.
- DC.L — Four bytes are allocated for numeric expressions. ASCII strings are right aligned on a 4-byte boundary.

Example for DC.B:

```
000000 4142 4344   Label: DC.B "ABCDE"
000004 45
000005 0A0A 010A          DC.B %1010, @12, 1, $A
000009 xx                 DC.B PAGE(Label)
```

Example for DC.W:

```
000000 0041 4243   Label: DC.W "ABCDE"
000004 4445
000006 000A 000A          DC.W %1010, @12, 1, $A
00000A 0001 000A
00000E xxxx               DC.W Label
```

Example for DC.L:

```
000000 0000 0041   Label: DC.L "ABCDE"
000004 4243 4445
000008 0000 000A         DC.L %1010, @12, 1, $A
00000C 0000 000A
000010 0000 0001
000014 0000 000A
000018 xxxx xxxx          DC.L Label
```

If the value in an operand expression exceeds the size of the operand, the value is truncated and a warning message is generated.

8.9 DCB — Define Constant Block

Syntax: [<label>:] `DCB` [<size>] <count>, <value>

where
<size> = `B` (default), `W`, or `L`

Description: The `DCB` directive causes the assembler to allocate a memory block initialized with the specified <value>. The length of the block is <size> * <count>.

<count> may not contain undefined, forward, or external references. It may range from 1 to 4096.

The value of each storage unit allocated is the sign-extended expression <value>, which may contain forward references. The <count> cannot be relocatable. This directive does not perform alignment.

These rules apply to size specifications for `DCB` directives:

- `DCB.B` — One byte is allocated for numeric expressions.
- `DCB.W` — Two bytes are allocated for numeric expressions.
- `DCB.L` — Four bytes are allocated for numeric expressions.

Example:

```
000000 FFFF FF     Label: DCB.B 3, $FF
000003 FFFE FFFE          DCB.W 3, $FFFE
000007 FFFE
000009 0000 FFFE          DCB.L 3, $FFFE
00000D 0000 FFFE
000011 0000 FFFE
```

8.10 DS — Define Space

Syntax: [<label>:] DS [.<size>] <count>

where
<size> = B (default), W, or L

Description: The DS directive is used to reserve memory for variables. Contents of the reserved memory is not initialized. The length of the block is <size> * <count>.

<count> may not contain undefined, forward, or external references. It may range from 1 to 4096.

Example:

```
Counter: DS.B  2  ; 2 contiguous bytes in memory
         DS.B  2  ; 2 contiguous bytes in memory
                  ; can only be accessed through the label Counter
         DS.L  5  ; 5 contiguous long words in memory
```

The label, Counter, references the lowest address of the defined storage area.

8.11 ELSE — Conditional Assembly

Syntax:

```
IF <condition>
[<assembly language statements>]
[ELSE]
<assembly language statements>]
ENDIF
```

Description: If <condition> is true, the statements between `IF` and the corresponding `ELSE` directive generate code.

If <condition> is false, the statements between `ELSE` and the corresponding `ENDIF` directive generate code. Nesting of conditional blocks is allowed. The maximum level of nesting is limited by the available memory at assembly time.

Example: The following is an example of the use of conditional assembly directives:

```
  Try: EQU 1
       IF  Try != 0
         LDA   #103
       ELSE
         LDA   #0
       ENDIF
```

The value of `Try` determines the instruction which generates code. As shown, the `LDA    #103` instruction generates code. Changing the operand of the equ directive to 0, causes the `LDA #0` instruction to generate code instead.

This listing is provided by the assembler for these lines of code:

```
Motorola HC05-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997

Abs. Rel.   Loc.   Obj. code   Source line
---- ----   ------ ---------   -----------
  1    1           0000 0001   Try: EQU 1
  2    2           0000 0001        IF  Try != 0
  3    3  000000 A667                LDA   #103
  4    4                           ELSE
  6    6                           ENDIF
```

8.12 END — End Assembly

Syntax: `END`

Description: The `END` directive indicates the end of the source code. Subsequent source statements in this file are ignored. The `END` directive in included files skips only subsequent source statements in the included file. Assembly continues in the included file.

Example: When assembling the code:

```
  Label: NOP
         NOP
         NOP
         END

         NOP ; No code generated
         NOP ; No code generated
```

The generated listing file is:

```
Motorola HC05-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997

Abs. Rel.   Loc.   Obj. code  Source line
---- ----   ------ ---------   -----------
1     1   000000 9D          Label: NOP
2     2   000001 9D                 NOP
3     3   000002 9D                 NOP
```

8.13 ENDIF — End Conditional Assembly

Syntax: `ENDIF`

Description: The `ENDIF` directive indicates the end of a conditional block. Nesting of conditional blocks is allowed. The maximum level of nesting is limited by the memory available at assembly time.

Example: See **8.18 IF — Conditional Assembly** for an example of `IF` directive.

8.14 ENDM — End Macro Definition

Syntax: `ENDM`

Description: The `ENDM` directive terminates both the macro definition and the macro expansion.

Example:

```
  5    5                          cpChar:  MACRO
  6    6                                     LDA \1
  7    7                                     STA \2
  8    8                                   ENDM
  9    9                          CodeSec: SECTION
 10   10                          Start:
 11   11                                   cpChar char1, char2
 12    6m  000000 C6 xxxx    +               LDA char1
 13    7m  000003 C7 xxxx    +               STA char2
 14   12   000006 9D                       NOP
 15   13   000007 9D                       NOP
```

8.15 EQU — Equate Symbol Value

Syntax: <label>: EQU <expression>

Description: The EQU directive assigns the value of the <expression> in the operand field to <label>. The <label> and <expression> fields are both required, and the <label> cannot be defined anywhere else in the program. The <expression> cannot include a symbol that is undefined.

The EQU directive does not allow forward references.

Example:

```
          0000 0014  MaxElement: EQU  20
          0000 0050  MaxSize:    EQU  MaxElement * 4

000000               Time:   DS.W 3
          0000 0000  Hour:   EQU Time  ; first word addr
          0000 0002  Minute: EQU Time+2; second word addr
          0000 0004  Second: EQU Time+4; third word addr
```

8.16 EVEN — Force Word Alignment

Syntax: `EVEN`

Description: This directive forces the next instruction to the next even address relative to the start of the section. `EVEN` is an abbreviation for `ALIGN 2`. Some processors require word and longword operations to begin at even address boundaries. In such cases, use of the `EVEN` directive ensures correct alignment, omission of the directive can result in an error message.

Example:

```
 6    6   000000                  ds.w  2
 ; location count has an even value, no padding byte inserted.
 7    7                           even
 8    8   000004                  ds.b  1
 ; location count has an odd value, one padding byte inserted.
 9    9   000005 00               even
10   10   000006                  ds.b  3
 ; location count has an odd value, one padding byte inserted.
11   11   000009 00               even
12   12          0000 000A aaa:   equ   10
```

8.17 FAIL — Generate Error Message

Syntax: `FAIL` <arg> | <string>

Description: Handling from the `FAIL` directive depends on the operand specified. If <arg> is in the range [500–$FFFFFFFF], the assembler generates a warning message, including the line number and argument of the directive.

Example: The following code segment:

```
cpChar: MACRO
          IFC "\1", ""
            FAIL 200
            MEXIT
          ELSE
            LDA \1
          ENDIF

          IFC "\2", ""
            FAIL 600
          ELSE
            STA \2
          ENDIF
        ENDM
codSec: SECTION
Start:
        cpChar char1
```

Generates this error message:

```
>> in "C:\MCUEZ\DEMO\WMMDS05A\test.asm", line 13, col 19, pos 226
          IFC "\2", ""
            FAIL 600
                   ^
WARNING A2332: FAIL found
  Macro Call : FAIL 600
```

If <arg> is in the range [0–499], the assembler generates an error message, including the line number and argument of the directive. The assembler does not generate an object file.

This code segment:

```
cpChar: MACRO
          IFC "\1", ""
            FAIL 200
            MEXIT
          ELSE
            LDA \1
          ENDIF

          IFC "\2", ""
            FAIL 600
          ELSE
            STA \2
          ENDIF
        ENDM
codSec: SECTION
Start:
        cpChar ,char2
```

Generates this error message:

```
>> in "C:\MCUEZ\DEMO\WMMDS05A\test.asm", line 6, col 19, pos 96

          IFC "\1", ""
            FAIL 200
                   ^
ERROR A2329: FAIL found
Macro Call : FAIL 200
```

If a string is supplied as the operand, the assembler generates an error message, including the line number and <string>. The assembler does not generate an object file.

Example:

```
cpChar: MACRO
          IFC "\1", ""
            FAIL "A character must be specified as first parameter"
            MEXIT
          ELSE
            LDA \1
          ENDIF
          IFC "\2", ""
            FAIL 600
          ELSE
            STA \2
          ENDIF
     ENDM
codeSec: SECTION
Start:
         cpChar , char2
```

Generates this error message:

```
>> in "C:\MCUEZ\DEMO\WMMDS05A\test.asm", line 7, col 17, pos 110
          IFC "\1", ""
            FAIL "A character must be specified as first parameter"
                   ^
ERROR A2338: A character must be specified as first parameter
Macro Call : FAIL "A character must be specified as first parameter"
```

The `FAIL` directive is intended for use with conditional assembly to detect a user-defined error or warning condition.

8.18 IF — Conditional Assembly

Syntax:

```
IF <condition>
[<assembly language statements>]
[ELSE]
[<assembly language statements>]
ENDIF
```

Description: If <condition> is true, the statements immediately following the IF directive generate code. Assembly continues until the corresponding `ELSE` or `ENDIF` directive is reached. Then all statements until the corresponding `ENDIF` directive are ignored. Nesting of conditional blocks is allowed. The maximum level of nesting is limited by available memory at assembly time.

The expected syntax for <condition> is:

```
<condition>    := <expression> <relation> <expression>
<relation>     :=  "=" | "!=" | " >=" | ">" | "<=" | "<" | "<>"
```

The <expression> must be absolute. It must be known at assembly time.

Example: This is an example of using conditional assembly directives:

```
Try: EQU 0
     IF  Try != 0
       LDA   #103
     ELSE
       LDA   #0
     ENDIF
```

The value of TRY determines the instruction that generates code. As shown, the LDA #0 instruction generates some code. Changing the operand of the EQU directive to one causes the LDA #103 instruction to generate code instead. This example shows the listing provided by the assembler for these lines of code:

```
1    1          0000 0000    Try: EQU 0
2    2          0000 0000         IF  Try != 0
4    4                            ELSE
5    5    000000 A600               LDA   #0
6    6                            ENDIF
```

8.19 IFCC — Conditional Assembly

Syntax: IFCC <condition>
[<assembly language statements>]
[ELSE]
[<assembly language statements>]
ENDIF

Description: These directives can be replaced by the IF directive. If IFCC <condition> is true, the statements immediately following the IFCC directive are assembled. Assembly continues until the corresponding ELSE or ENDIF directive is reached, after which, assembly moves to the statements following the ENDIF directive. Nesting of conditional blocks is allowed. The maximum level of nesting is limited by the memory available at assembly time.

Table 8-9 lists the available conditional types.

Table 8-9. Conditional Types

IFCC	Condition	Meaning
IFEQ	<expression>	IF <expression> == 0
IFNE	<expression>	IF <expression> != 0
IFLT	<expression>	IF <expression> < 0
IFLE	<expression>	IF<expression> <= 0
IFGT	<expression>	IF <expression> > 0
IFGE	<expression>	IF <expression> >= 0
IFC	<string1>, <string2>	IF <string1> == <string2>
IFNC	<string1>, <string2>	IF <string1> != <string2>
IFDEF	<label>	IF <label> was defined
IFNDEF	<label>	IF <label> was not defined

Example: This is an example use of conditional assembly directives:

```
Try: EQU 0
     IFNE  Try
       LDA #103
      ELSE
       LDA #0
      ENDIF
```

The value of TRY determines the instruction to be assembled in the program. As shown, the LDA #0 instruction generates some code. Changing the directive to IFEQ causes the LDA #103 instruction to generate code instead. This example shows the listing provided by the assembler for these lines of code:

```
1    1          0000 0000   Try: EQU 0
2    2          0000 0000        IFNE  Try
4    4                           ELSE
5    5   000000 A600               LDA #0
6    6                           ENDIF
```

8.20 INCLUDE — Include Text from Another File

Syntax: `INCLUDE` <file specification>

Description: This directive causes the included file to be inserted in the source input stream. The <file specification> is not case sensitive and must be enclosed in quotation marks.

The assembler attempts to open <file specification> relative to the current working directory. If the file is not found, then it is searched for in each path specified in the environment variable GENPATH.

Example:

```
INCLUDE "..\LIBRARY\macros.inc"
```

8.21 LIST — Enable Listing

Syntax `LIST`

Description: Specifies that the instructions which follow must be inserted in the listing and debug files. The listing file is generated only if the option `-L` is specified on the command line.

The source text following the `LIST` directive is listed until a `NOLIST` or an `END` is reached.

This directive is not written to the listing and debug file. When neither the `LIST` nor the `NOLIST` directives are specified in a source file, all instructions are written to the list file.

Example: This portion of code:

```
aaa:    nop

        list
bbb:    nop
        nop

        nolist
ccc:    nop
        nop

        list

ddd:    nop
        nop
```

Generates this listing file:

```
Motorola HC05-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997

Abs. Rel.   Loc.   Obj. code   Source line
---- ----   ------ ---------   -----------
 1    1   000000 9D          aaa:    nop
 2    2
 4    4   000001 9D          bbb:    nop
 5    5   000002 9D                  nop
 6    6
12   12
13   13   000005 9D          ddd:    nop
14   14   000006 9D                  nop
15   15
```

The gap in the location counter is due to instructions defined inside the NOLIST-LIST block.

See also: **8.27 NOLIST — Disable Listing**

8.22 LLEN — Set Line Length

Syntax: `LLEN` <n>

Description: Sets the number of characters, <n>, from the source line that are included on the listing line. The values allowed for <n> are in the range [0–132]. If a value smaller than 0 is specified, the line length is set to 0. If a value bigger than 132 is specified, the line length is set to 132.

Lines of the source file that exceed the specified number of characters are truncated in the listing file.

Example: This portion of code:

```
dc.b      5
llen      $20
dc.w      $4567, $2345
llen      $17
dc.w      $4567, $2345
even
nop
```

Generates this listing file:

```
 Motorola HC05-Assembler
 (c) COPYRIGHT MOTOROLA 1991-1997

 Abs. Rel.   Loc.   Obj. code       Source line
 ---- ----   ------ ---------       -----------
 1    1   000000 05                  dc.b      5
 3    3
 4    4   000001 4567 2345           dc.w      $4567, $2345
 5    5
 7    7   000005 4567 2345           dc.w      $4567
 8    8   000009 00                  even
 9    9   00000A 9D                  nop
10   10
```

The `LLEN` $17 directive causes the second `dc.w $4567, $2345` to be truncated in the assembler listing file. The generated code is correct.

8.23 LONGEVEN — Forcing Longword Alignment

Syntax: `LONGEVEN`

Description: This directive forces the next instruction to the next long-word address relative to the start of the section. `LONGEVEN` is an abbreviation for `ALIGN 4`.

Example:

```
  2    2   000000 01                    dcb.b 1,1
          ; location counter is not a multiple of 4, 3 filling bytes
          ; are required.
  3    3   000001 0000 00               longeven
  4    4   000004 0002 0002             dcb.w 2,2
          ; location counter is already a multiple of 4, no filling
          ; bytes are required.
  5    5                                longeven
  6    6   000008 0202                  dcb.b 2,2
  7    7  ; following is for text section
  8    8                       s27         SECTION 27
  9    9   000000 A7                    nop
          ; location counter is not a multiple of 4, 3 filling bytes
          ; are required.
 10   10   000001 0000 00               longeven
 11   11   000004 A7                    nop
```

8.24 MACRO — Begin Macro Definition

Syntax: <label>: `MACRO`

Description: The <label> of the MACRO directive is the name by which the macro is called. This name must not be a processor machine instruction or assembler directive name. For more information on macros, refer to **Section 9. Macros**.

Example:

```
Motorola HC05-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997

Abs. Rel.   Loc.   Obj. code   Source line
---- ----   ------ ---------   -----------
 1    1                                  XDEF Start
 2    2                         MyData:  SECTION
 3    3     000000                char1:   DS.B  1
 4    4     000001                char2:   DS.B  1
 5    5                         cpChar:  MACRO
 6    6                                    LDA \1
 7    7                                    STA \2
 8    8                                  ENDM
 9    9                         CodeSec: SECTION
10   10                         Start:
11   11                                  cpChar char1, char2
12    6m    000000 C6 xxxx    +              LDA char1
13    7m    000003 C7 xxxx    +              STA char2
14   12     000006 9D                      NOP
15   13     000007 9D                      NOP
16   14     000008 9D                      NOP
17   15
```

8.25 MEXIT — Terminate Macro Expansion

Syntax: `MEXIT`

Description: `MEXIT` is usually used together with conditional assembly within a macro. In that case, the macro expansion might terminate prior to termination of the macro definition. The `MEXIT` directive causes macro expansion to skip any remaining source lines ahead of the `ENDM` directive.

Example: This portion of code:

```
          XDEF  entry
storage: EQU $00FF
save:     MACRO           ; Start macro definition
           LDHX #storage
           LDA  \1
           STA  0,x       ; save first argument
           LDA  \2
           STA  2,x       ; save second argument
           IFC  '\3', ''  ; is there a 3rd argument ?
             MEXIT        ; no, exit from macro.
           ENDC
           LDA  \3        ;save third argument
           STA  4,X
          ENDM            ; End of macro definition
codSec:   SECTION
entry:
          save char1, char2
```

Generates this listing file:

```
Motorola HC05-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997

Abs. Rel.   Loc.   Obj. code     Source line
---- ----   ------ ---------     -----------
 1    1                                 XDEF  entry
 2    2          0000 00FF   storage: EQU $00FF
 3    3
 4    4                      save:    MACRO    ; Start macro definition
 5    5                               LDX  #storage
 6    6                               LDA  \1
 7    7                              STA  0,x ; save first argument
 8    8                               LDA  \2
 9    9                            STA  2,x    ; save second argument
 10   10                               IFC  '\3', ''; is there a 3rd
 11   11                                MEXIT ; no, exit from macro.
 12   12                               ENDC
 13   13                               LDA  \3 ;save third argument
 14   14                               STA  4,X
 15   15                             ENDM      ;End of macro definition
 16   16
 17   17                       datSec:  SECTION
 18   18    000000             char1:   ds.b 1
 19   19    000001             char2:   ds.b 1
 20   20
 21   21
 22   22
 23   23                       codSec:  SECTION
 24   24                       entry:
 25   25                                save char1, char2
 26    5m   000000 AEFF      +          LDX  #storage
 27    6m   000002 C6 xxxx   +          LDA  char1
 28    7m   000005 E700      +       STA  0,x    ; save first argument
 29    8m   000007 C6 xxxx   +          LDA  char2
 30    9m   00000A E702      +          STA  2,x    ; save second
 31   10m          0000 0001 +          IFC  '', ''; is there a 3rd
 33   11m                    +         MEXIT       ;no,exit from macro.
 34   12m                    +          ENDC
 35   13m                  +         LDA           ;save third argument
 36   14m                    +          STA  4,X
```

8.26 MLIST — List Macro Expansions

Syntax: `MLIST` [ON | OFF]

Description: When `ON` is entered with an `MLIST` directive, the assembler includes the macro expansions in the listing and debug files. When `OFF` is entered, macro expansions are omitted from the listing and debug files. This directive is not written to the listing and debug file, and the default value is `ON`.

Example: The following listing shows a macro definition and expansion with `MLIST ON`:

```
        XDEF  entry
        MLIST ON
swap:   MACRO
          LDA   \1
          LDX   \2
          STA   \2
          STX   \1
        ENDM
codSec: SECTION
entry:
        LDA   #$F0
        LDX   #$0F
main:
        STA   first
        STX   second
        swap  first, second
        NOP
        BRA   main
datSec: SECTION
first:  DS.B  1
second: DS.B  1
```

The assembler listing file is:

```
 1    1                                 XDEF  entry
 3    3                         swap:   MACRO
 4    4                                   LDA   \1
 5    5                                   LDX   \2
 6    6                                   STA   \2
 7    7                                   STX   \1
 8    8                                 ENDM
 9    9
10   10                         codSec: SECTION
11   11                         entry:
12   12    000000 A6F0                  LDA   #$F0
13   13    000002 AE0F                  LDX   #$0F
14   14                         main:
15   15    000004 C7 xxxx               STA   first
16   16    000007 CF xxxx               STX   second
17   17                                 swap  first, second
18    4m   00000A C6 xxxx     +           LDA   first
19    5m   00000D CE xxxx     +           LDX   second
20    6m   000010 C7 xxxx     +           STA   second
21    7m   000013 CF xxxx     +           STX   first
22   18    000016 9D                    NOP
23   19    000017 20EB                  BRA   main
24   20
25   21                         datSec: SECTION
26   22    000000               first:  DS.B  1
27   23    000001               second: DS.B  1
```

For the same code, with `MLIST OFF`, the listing file is:

```
 Motorola HC05-Assembler
 (c) COPYRIGHT MOTOROLA 1991-1997

 Abs. Rel.   Loc.   Obj. code   Source line
 ---- ----   ------ ---------   -----------
 1    1                                 XDEF  entry
 3    3                         swap:   MACRO
 4    4                                   LDA   \1
 5    5                                   LDX   \2
 6    6                                   STA   \2
 7    7                                   STX   \1
 8    8                                 ENDM
 9    9                         codSec: SECTION
10   10                         entry:
11   11    000000 A6F0                  LDA   #$F0
12   12    000002 AE0F                  LDX   #$0F
13   13                         main:
14   14    000004 C7 xxxx               STA   first
15   15    000007 CF xxxx               STX   second
16   16                                 swap  first, second
21   17    000016 9D                    NOP
22   18    000017 20EB                  BRA   main
23   19                         datSec: SECTION
24   20    000000               first:  DS.B  1
25   21    000001               second: DS.B  1
```

8.27 NOLIST — Disable Listing

Syntax: `NOLIST`

Description: Suppresses printing of the following instructions in the assembly listing and debug files until a `LIST` directive is reached.

Example: This portion of code:

```
  aaa:     nop
           list
  bbb:     nop
           nop
           nolist
  ccc:     nop
           nop
           list
  ddd:     nop
           nop
```

Generates this listing file:

```
 1     1    000000 9D           aaa:    nop

 3     3    000001 9D           bbb:    nop

 4     4    000002 9D                   nop

 9     9    000005 9D           ddd:    nop

10    10    000006 9D                   nop

11    11
```

The gap in the location counter is due to instructions defined inside a `NOLIST` block.

8.28 NOPAGE — Disable Paging

Syntax: `NOPAGE`

Description: Disables pagination in the listing file. Program lines are listed continuously without headings or top or bottom margins.

8.29 ORG — Set Location Counter

Syntax: `ORG` <expression>

Description: The `ORG` directive sets the location counter to the value specified by <expression>. Subsequent statements are assigned memory locations starting with the new location counter value. The <expression> must be absolute and may not contain any forward, undefined, or external references. The `ORG` directive generates an internal section, which is absolute.

Example:

```
        org    $2000
b1:     nop
b2:     rts
```

8.30 OFFSET — Create Absolute Symbols

Syntax: `OFFSET` <expression>

Synonym: None

Description: The `OFFSET` directive declares an offset section and initializes the location counter to the value specified in <expression>. The <expression> must be absolute and cannot contain references to external, undefined, or forward defined labels.

The `OFFSET` section is useful to simulate data structures or stack frames.

Example: The following example shows how `OFFSET` can be used to access elements of a structure.

```
  1    1                              ; Test file for OFFSET
  2    2
  3    3                               XDEF entry
  4    4          0000 0C00   StackInit: EQU $C00
  5    5
  6    6                              OFFSET 0
  7    7   000000             ID:      DS.B   1
  8    8   000001             COUNT:   DS.W   1
  9    9   000003             VALUE:   DS.L   1
 10   10          0000 0007   SIZE:    EQU *
 11   11
 12   12
 13   13                      DataSec: SECTION
 14   14   000000             Struct:    DS.B SIZE
 15   15
 16   16                      CodeSec: SECTION
 17   17                      entry:
 18   18                              ;LDHX  #StackInit
 19   19                              ;TXS
 20   20                              ;LDHX  #Struct
 21   21   000000 A600               LDA   #0
 22   22   000002 E700               STA   ID, X
 23   23   000004 6C01               INC   COUNT, X
 24   24   000006 4C                 INCA
 25   25   000007 E703               STA   VALUE, X
 26   26
 27   27                      return:
```

When a statement affecting the location counter (other than `EVEN`, `LONGEVEN`, `ALIGN`, or `DS`) is encountered after the `OFFSET` directive, the offset section is ended. The preceding section is activated again, and the location counter is restored to the next available location in this section.

8.31 PAGE — Insert Page Break

Syntax: `PAGE`

Description: Inserts a page break in the assembly listing

Example: This portion of code:

```
  codeSec: SECTION
           nop
           nop
           page
           nop
           nop
```

Generates this listing file:

```
Motorola HC05-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997

Abs. Rel.   Loc.   Obj. code   Source line
---- ----   ------ ---------   -----------
1    1                       codeSec: SECTION
2    2   000000 9D                     NOP
3    3   000001 9D                     NOP

Motorola HC05-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997

Abs. Rel.   Loc.   Obj. code   Source line
---- ----   ------ ---------   -----------
5    5   000002 9D                     NOP
6    6   000003 9D                     NOP
```

8.32 PLEN — Set Page Length

Syntax:	`PLEN` <n>
Description:	Sets the listings page length to <n> lines. <n> may range from 10 to 10,000. If the number of lines already listed on the current page is greater than or equal to <n>, listing will continue on the next page with the new page length setting. The default page length is 65 lines.

8.33 SECTION — Declare Relocatable Section

Syntax: <name>: `SECTION [SHORT]`[<number>]

Description: This directive declares a relocatable section and initializes the location counter for the following code. The first `SECTION` directive for a section sets the location counter to 0. Subsequent `SECTION` directives for that section restore the location counter to the value that follows the address of the last code in the section. A section is a code section when it contains at least an assembly instruction. It is considered to be a constant section if it contains only `DC` or `DCB` directives. A section is considered to be a data section if it contains at least a DS directive or if it is empty.

<name> is the name assigned to the section.

<number> is optional and is specified only for compatibility with the MASM assembler.

Example: The following example defines a section `aaa`, which is split into two blocks with section `bbb` between them. The location counter associated with label zz is 1, because a NOP instruction was already defined in this section at label xx.

```
2    2                          aaa:    section 4
3    3   000000 9D              xx:     nop
4    4                          bbb:    section 5
5    5   000000 9D              yy:     nop
6    6   000001 9D                      nop
7    7   000002 9D                      nop
8    8                          aaa:    section 4
9    9   000001 9D              zz:     nop
```

The optional qualifier `SHORT` specifies that the section is a short section. Objects defined there can be accessed using the direct addressing mode.

Example: The following example demonstrates the definition and usage of a `SHORT` section. On line number 12, the symbol data is accessed using the direct addressing mode.

```
1    1                          dataSec: SECTION  SHORT
2    2   000000                 data:    DS.B 1
3    3
4    4                          codeSec: SECTION
5    5
6    6                          entry:
7    7   000000 9C                       RSP
8    8   000001 A600                     LDA #0
9    9   000003 B7xx                     STA data
```

8.34 SET — Set Symbol Value

Syntax: <label>: SET <expression>

Description: Similar to the EQU directive, the SET directive assigns the value of the <expression> in the operand field to the symbol in the <label> field. The <expression> cannot include a symbol that is undefined or not yet defined. The <label> is an assembly time constant, SET does not generate any machine code.

The value is temporary; a subsequent SET directive can redefine it.

Example:

```
Motorola HC05-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997

Abs. Rel.   Loc.   Obj. code   Source line
---- ----   ------ ---------   -----------
 1    1          0000 0002   count: SET 2
 2    2   000000 02          loop:  DC.B count
 3    3          0000 0002          IFNE count
 4    4          0000 0001   count: SET count - 1
 5    5                             ENDIF
 6    6   000001 01                 DC.B count
 7    7          0000 0001          IFNE count
 8    8          0000 0000   count: SET count - 1
 9    9                             ENDIF
10   10   000002 00                 DC.B count
11   11          0000 0000          IFNE count
12   12                             ENDIF
```

The value associated with the label count is decremented after each DC.B instruction.

8.35 SPC — Insert Blank Lines

Syntax:	SPC <count>
Description:	Inserts blank lines in the assembly listing. <count> may range from 0 to 65. This has the same effect as manually entering blank lines in the assembly source. A blank line is a line containing only a carriage return.

8.36 TABS — Set Tab Length

Syntax:	TABS <n>
Description:	Sets tab length to <n> spaces. The default tab length is eight. <n> may range from 0 to 128.

8.37 TITLE — Provide Listing Title

Syntax:	`TITLE "title"`
Description:	Prints the <title> on the head of each page of the listing file. This directive must be the first source code line. A title consists of a string of characters enclosed in quotes (").

The title will be written at the top of each page in the assembly listing file.

For compatibility purposes with MASM, a title also can be specified without quotes.

8.38 XDEF — External Symbol Definition

Syntax: `XDEF` [.<size>] <label>[,<label>]...

where
<size> = `B`, `W` (default), or `L`

Description: This directive specifies labels defined in the current module that are to be passed to the linker as labels that can be referenced by other modules linked to the current module.

The number of symbols listed in an `XDEF` directive is limited by the memory available at assembly time.

Example:

```
XDEF Global      ; Global can be referenced in other module
XDEF AnyCase     ; Note that the linker and assembler are
                 ; case sensitive to names.

GLOBAL: DS.B 4
...
AnyCase NOP
```

8.39 XREF — External Symbol Reference

Syntax: `XREF` [.<size>] <symbol>[,<symbol>]...

where
<size> = `B, W` (default), or `L`

Description: This directive specifies symbols referenced in the current module but defined in another module. The list of symbols and corresponding 16-bit values are passed to the linker.

The number of symbols listed in an `XREF` directive is limited by the memory available at assembly time.

Example:

```
XREF OtherGlobal; Reference "OtherGlobal" defined in another
               ; module
```

See also: **8.38 XDEF — External Symbol Definition**)

Section 9. Macros

9.1 Contents

9.2 Introduction 195
9.3 Macro Overview 195
9.4 Defining a Macro 196
9.5 Calling Macros 197
9.6 Macro Parameters 197
9.7 Labels Inside Macros 198
9.8 Macro Expansion 199
9.9 Nested Macros 200

9.2 Introduction

This section describes how macros are used with the MCUez HC05 assembler.

9.3 Macro Overview

A macro is a template for a code sequence. Once a macro is defined, subsequent references to the macro name are replaced by its code sequence. A macro must be defined before it is called. When a macro is defined, it is given a name. This name becomes the mnemonic by which the macro is subsequently called.

The assembler expands the macro definition each time the macro is called. The macro call causes generation of source statements, which may include macro arguments. A macro definition may contain any code or directive except nested macro definitions. Calling previously defined macros is also allowed. Source statements generated by a macro call are inserted in the source file at the position where the macro is invoked.

To call a macro, enter the macro name in the operation field of a source statement. Place the arguments in the operand field. The macro may contain conditional assembly directives that cause the assembler to produce inline coding variations of the macro definition.

Macro calls produce inline code to perform a predefined function. Each time the macro is called, code is inserted in the normal flow of the program so that the generated instructions are executed in line with the rest of the program.

9.4 Defining a Macro

The definition of a macro consists of four parts:

1. The header statement, a `MACRO` directive with a label that names the macro
2. The body of the macro, a sequential list of assembler statements, some possibly including argument placeholders
3. The `ENDM` directive, which terminates the macro definition
4. The `MEXIT` directive, which stops macro expansion

The body of a macro is a sequence of assembler source statements. Macro parameters are defined by the appearance of parameter designators within these source statements. Valid macro definition statements include assembly language instructions, assembler directives, and calls to previously defined macros. However, macro definitions may not be nested.

NOTE: *Refer to **Section 8. Assembler Directives** for information about the `MACRO`, `ENDM`, `MEXIT`, and `MLIST` directives.*

9.5 Calling Macros

The form of a macro call is:

[<label>:] <name>[.<sizearg>] [<argument> [,<argument>]...]

Although a macro may be referenced by another macro prior to its definition in the source module, all macros must be defined before their first call. The name of the called macro must appear in the operation field of the source statement. Arguments are supplied in the operand field of the source statement, separated by commas.

The macro call produces inline code at the location of the call, according to the macro definition and the arguments specified in the macro call. The source statements of the expanded macro then are assembled subject to the same conditions and restrictions affecting any source statement. Nested macro calls also are expanded at this time.

9.6 Macro Parameters

As many as 36 different substitutable parameters can be used in the source statements that constitute the body of a macro. These parameters are replaced by the corresponding arguments in a subsequent call to that macro.

A parameter designator consists of a back slash character (\), followed by a digit (0– 9) or an upper case letter (A–Z). Parameter designator \0 corresponds to a size argument that follows the macro name, separated by a period (.).

Example: Consider this macro definition:

```
MyMacro: MACRO
         DC.\0   \1, \2
         ENDM
```

When this macro is used in a program, for instance:

```
MyMacro.B $10, $56
```

The assembler expands it to:

```
  DC.B $10, $56
```

Arguments in the operand field of the macro call refer to parameter designators \1 through \9 and \A through \Z, in that order. The argument list (operand field) of a macro call cannot be extended onto additional lines.

At the time of a macro call, arguments from the macro call are substituted as literal (string) substitutions for parameter designators in the body of the macro. The string corresponding to a given argument is substituted literally wherever that parameter designator occurs in a source statement as the macro is expanded. Each statement generated in the execution is assembled inline.

It is possible to specify a null argument in a macro call by placing a comma immediately after a macro name. There should be no spaces or characters between the comma and macro name or comma that follows an argument. When a null argument is passed as an argument in a nested macro call, a null value is passed. All arguments have a default value of null at the time of a macro call.

9.7 Labels Inside Macros

To avoid the problem of multiple-defined labels resulting from multiple calls to a macro that has labels in its source statements, the programmer can direct the assembler to generate unique labels on each macro.

Assembler-generated labels include a string of the form _nnnnn where nnnnn is a 5-digit value. The programmer requests an assembler-generated label by specifying \@ in a label field within a macro body. Each successive label definition that specifies a \@ directive generates a successive value of _nnnnn, thereby creating a unique label on each macro call. Note that \@ may be preceded or followed by additional characters for clarity and to prevent ambiguity.

Example:

```
clear: MACRO
         LDX    \1
         LDA    #16
\@LOOP:  CLR    0,X
         INCX
         DECA
         BNE    \@LOOP
       ENDM
```

This macro is called in the application:

```
          clear     temporary
          clear     data
```

The two macro calls of `clear` are expanded this way:

```
          clear  temporary
              LDX    temporary
              LDA    #16
    _00001LOOP:CLR    0,X
              INCX
              DECA
              BNE    _00001LOOP
         clear   data
              LDX    #data
              LDA    #16
    _00002LOOP:CLR    0,X
              INCX
              DECA
              BNE   _00002LOOP
```

9.8 Macro Expansion

When the assembler reads a statement in a source program that calls a previously defined macro, it processes the call as described here.

The symbol table is searched for the macro name. If it is not in the symbol table, an undefined symbol error message is issued.

The rest of the line is scanned for arguments. Any argument in the macro call is saved as a literal or null value in one of the 35 possible parameter fields. When the number of arguments in the call is less than the number of parameters used in the macro, the arguments (which have not been defined at invocation time) are initialized with an empty string (“ “).

Starting with the line following the `MACRO` directive, each line of the macro body is saved and associated with the named macro. Each line is retrieved in turn, with parameter designators replaced by argument strings or assembler-generated label strings.

Once the macro is expanded, the source lines are evaluated and object code is produced.

9.9 Nested Macros

Macro expansion is performed at invocation, which is also the case for nested macros. If the macro definition contains a nested macro call, the nested macro expansion takes place inline. Recursive macro calls are also supported.

A macro call is limited to the length of one line, which is 1024 characters.

Section 10. Assembler Listing File

10.1 Contents

10.2 Introduction 201

10.3 Page Header 202

10.4 Source Listing 202
10.4.1 Absolute (Abs.) Listing 203
10.4.2 Relative (Rel.) Listing 204
10.4.3 Location (Loc.) Listing 205
10.4.4 Object (Obj.) Code 206
10.4.5 Source Line 207

10.2 Introduction

The assembler listing file is the output file that contains information about the generated code. The listing file is generated if the option –L is entered on the command line. If an error is detected during assembly, no listing file is generated.

The amount of information available depends on these assembly options:

-Li, -Lc, -Ld, -Le

Information in the listing file also depends on these assembly directives:

LIST, NOLIST, CLIST, MLIST

The format of the listing file is influenced by these directives:

PLEN, LLEN, TABS, SPC, PAGE, NOPAGE, TITLE

The name of the generated listing file is <base name>.*lst*

10.3 Page Header

The page header consists of three lines:

1. The first line contains an optional user string defined in the directive TITLE
2. The second line contains the vendor and target processor names (MOTOROLA/HC05)
3. The third line contains a copyright notice.

Example:

```
Demo Application
Motorola HC05-Assembler
(c) COPYRIGHT MOTOROLA 1991-1997
```

10.4 Source Listing

The source listing is divided into five columns:

1. Abs.
2. Rel.
3. Loc.
4. Obj. code
5. Source line

10.4.1 Absolute (Abs.) Listing

This column contains the absolute line number for each instruction. The absolute line number is the line number in the `DBG` file, which contains all included files and where all macro calls have been expanded.

Example:

```
Abs. Rel.   Loc.    Obj. code   Source line
---- ----   ------  ---------   -----------
  1    1                       ;------------------------------
  2    2                       ; File: test.o
  3    3                       ;------------------------------
  4    4
  5    5                                XDEF Start
  6    6                      MyData:   SECTION
  7    7   000000             char1:    DS.B  1
  8    8   000001             char2:    DS.B  1
  9    9                                INCLUDE "macro.inc"
 10    1i                     cpChar:   MACRO
 11    2i                                 LDA \1
 12    3i                                 STA \2
 13    4i                               ENDM
 14   10                      CodeSec:  SECTION
 15   11                      Start:
 16   12                                cpChar char1, char2
 17    2m  000000 C6 xxxx    +            LDA char1
 18    3m  000003 C7 xxxx    +            STA char2
 19   13   000006 9D                    NOP
 20   14   000007 9D                    NOP
```

10.4.2 Relative (Rel.) Listing

This column contains the relative line number for each instruction. The relative line number is the line number in the source file. For included files, the relative line number is the line number in the included file. For macro call expansion, the relative line number is the line number of the instruction in the macro definition.

An `i` suffix is appended to the relative line number when the line comes from an included file. An `m` suffix is appended to the relative line number when the line is generated by a macro call.

Example:

```
 Abs. Rel.   Loc.    Obj. code   Source line
 ---- ----  ------   ---------   -----------
 1     1                        ;------------------------------
 2     2                        ; File: test.o
 3     3                        ;------------------------------
 4     4
 5     5                                 XDEF Start
 6     6                     MyData:     SECTION
 7     7    000000           char1:      DS.B  1
 8     8    000001           char2:      DS.B  1
 9     9                                 INCLUDE "macro.inc"
10     1i                    cpChar:    MACRO
11     2i                                  LDA \1
12     3i                                  STA \2
13     4i                               ENDM
14    10                     CodeSec:   SECTION
15    11                     Start:
16    12                                 cpChar char1, char2
17     2m   000000 C6 xxxx    +              LDA char1
18     3m   000003 C7 xxxx    +              STA char2
19    13   000006 9D                     NOP
20    14   000007 9D                     NOP
```

In the previous example, the line number displayed in the column `Rel.` represents the line number of the corresponding instruction in the source file. `1i` on line number 10 denotes that the instruction `cpChar: MACRO` is located in an included file. `2m` on line number 17 denotes that the instruction `LDA char` is generated by a macro expansion.

10.4.3 Location (Loc.) Listing

This column contains the address of the instruction. For absolute sections, the address is preceded by the letter `a` and contains the absolute address of the instruction. For relocatable sections, this address is the offset of the instruction from the beginning of the relocatable section. This address is a hexadecimal number coded on 6 digits. A value is written in this column in front of each instruction generating code or allocating storage. This column is empty in front of each instruction that does not generate code (for example, SECTION, XDEF, etc.).

Example:

```
 Abs. Rel.   Loc.    Obj. code   Source line
 ---- ----   ------  ---------   -----------
  1    1                        ;------------------------------
  2    2                        ; File: test.o
  3    3                        ;------------------------------
  4    4
  5    5                               XDEF Start
  6    6                      MyData:  SECTION
  7    7    000000           char1:   DS.B  1
  8    8    000001           char2:   DS.B  1
  9    9                               INCLUDE "macro.inc"
 10    1i                     cpChar:  MACRO
 11    2i                                LDA \1
 12    3i                                STA \2
 13    4i                              ENDM
 14   10                      CodeSec: SECTION
 15   11                      Start:
 16   12                               cpChar char1, char2
 17    2m   000000 C6 xxxx    +         LDA char1
 18    3m   000003 C7 xxxx    +         STA char2
 19   13    000006 9D                 NOP
 20   14    000007 9D                 NOP
```

In the previous example, the hexadecimal number displayed in the column `Loc.` is the offset of each instruction in the section `codeSec`. There is no location counter specified in front of the instruction `INCLUDE "macro.inc"` because this instruction does not generate code. The instruction `LDA char1` is located at offset 0 from the `codeSec` section start address. The instruction `STA char2` is located at offset 3 from the `codeSec` section start address.

10.4.4 Object (Obj.) Code

This column contains the hexadecimal code of each instruction in hexadecimal format. This code is not identical to the code stored in the object file. The letter `x` is displayed at the position where the address of an external or relocatable label is expected. Code at the `x` position is determined at link time.

Example:

```
Abs. Rel.   Loc.     Obj. Code     Source Line
---- ----   ------   ---------     -----------
 1    1                         ;------------------------------
 2    2                         ; File: test.o
 3    3                         ;------------------------------
 4    4
 5    5                                 XDEF Start
 6    6                       MyData:   SECTION
 7    7     000000                 char1:   DS.B  1
 8    8     000001                 char2:   DS.B  1
 9    9                                 INCLUDE "macro.inc"
10    1i                      cpChar:   MACRO
11    2i                                  LDA \1
12    3i                                  STA \2
13    4i                                ENDM
14   10                       CodeSec:  SECTION
15   11                       Start:
16   12                                 cpChar char1, char2
17    2m    000000 C6 xxxx   +              LDA char1
18    3m    000003 C7 xxxx   +              STA char2
19   13     000006 9D                      NOP
20   14     000007 9D                   NOP
```

10.4.5 Source Line

This column contains the source statement. This is a copy of the source line from the source module. For lines resulting from a macro expansion, the source line is the expanded line as a result of parameter substitution.

Example:

```
Abs. Rel.   Loc.     Obj. code    Source line
---- ----   ------   ---------    -----------
 1    1                          ;-------------------------------
 2    2                          ; File: test.o
 3    3                          ;-------------------------------
 4    4
 5    5                                   XDEF Start
 6    6                          MyData:  SECTION
 7    7      000000                 char1:   DS.B  1
 8    8      000001                 char2:   DS.B  1
 9    9                                   INCLUDE "macro.inc"
10    1i                         cpChar:  MACRO
11    2i                                    LDA \1
12    3i                                    STA \2
13    4i                                  ENDM
14   10                          CodeSec: SECTION
15   11                          Start:
16   12                                   cpChar char1, char2
17    2m     000000 C6 xxxx    +        LDA char1
18    3m     000003 C7 xxxx    +        STA char2
19   13      000006 9D                      NOP
20   14      000007 9D                      NOP
```

Section 11. Operating Procedures

11.1 Contents

11.2 Introduction 210

11.3 Working with Absolute Sections 210
11.3.1 Defining Absolute Sections in the Assembly Source File 210
11.3.2 Linking an Application Containing Absolute Sections 212

11.4 Working with Relocatable Sections 213
11.4.1 Defining Relocatable Sections in the Assembly Source File 213
11.4.2 Linking an Application Containing Relocatable Sections 214

11.5 Initializing the Vector Table 216
11.5.1 Initializing the Vector Table in the Linker PRM File 217
11.5.2 Initializing Vector Table in Assembly Source Files Using a Relocatable Section 219
11.5.3 Initializing the Vector Table in the Assembly Source File Using an Absolute Section 222

11.6 Splitting an Application into Different Modules 225

11.7 Using Direct Addressing Mode to Access Symbols 227
11.7.1 Using Direct Addressing Mode to Access External Symbols 227
11.7.2 Using Direct Addressing Mode to Access Exported Symbols 227
11.7.3 Defining Symbols in the Direct Page 228
11.7.4 Using a Force Operator 228
11.7.5 Using SHORT Sections 229

11.8 Generating an .abs File Directly 229
11.8.1 Assembler Source File 230
11.8.2 Assembling and Generating the Application 231

11.2 Introduction

This section provides operating procedures for the MCUez assembler.

11.3 Working with Absolute Sections

An absolute section is a section in which the start address is known at assembly time. (See modules *fiboorg.asm* and *fiboorg.prm* in the demo directory.)

11.3.1 Defining Absolute Sections in the Assembly Source File

An absolute section is defined using the directive `ORG`. The macro assembler generates a pseudo section named `ORG_`<index>, where< index> is an integer that is incremented each time an absolute section is encountered.

Example:

Defining an absolute section containing data:

```
      ORG   $A00     ; Absolute constant data section
cst1: DC.B $A6
cst2: DC.B $BC
      ORG   $800     ; Absolute data section
var:  DS.B    1
```

In the previous portion of code, the label `cst1` will be located at address `$A00`, and label `cst2` will be located at address `$A01`, as shown in the next code listing.

```
1   1                       ORG   $A00 ; Absolute
2   2   a000A00 A6    cst1:    DC.B $A6
3   3   a000A01 BC    cst2:    DC.B $BC
4   4
5   5                       ORG   $800  ; Absolute data
6   6   a000800         var:    DS.B    1
```

Defining an absolute section containing code:

```
            ORG  $C00   ; Absolute code section
   entry:
            LDA  cst1   ; Load value in cst1
            ADD  cst2   ; Add value in cst2
            STA  var    ; Store in var
            BRA  entry
```

In the previous portion of code, the instruction `LDA` will be located at address `$C00`, and instruction `ADD` at address `$C03`, as shown in the code listing here.

```
  8    8                        ORG  $C00 ; Absolute code
  9    9                                  entry:
 10   10    a000C00 C6 0A00     LDA  cst1 ; Load value
 11   11    a000C03 CB 0A01     ADD  cst2 ; Add value
 12   12    a000C06 C7 0800     STA  var  ; Store in var
 13   13    a000C09 20F5        BRA  entry
 14   14
```

To avoid problems during linking or executing an application, an assembly file should at least:

- Initialize the stack pointer. The instruction `RSP` can be used to initialize the stack pointer to `$00FF`.
- Publish the application entry point using `XDEF`.

The programmer should ensure that the addresses specified in the source file are valid addresses for the MCU being used.

11.3.2 Linking an Application Containing Absolute Sections

Applications containing only absolute sections must be linked. The linker parameter file must contain at least:

- Name of the absolute file
- Name of the object file that should be linked.The specification of a memory area where the sections containing variables must be allocated. For applications containing only absolute sections, nothing will be allocated there.
- Specification of a memory area where the sections containing code or constants must be allocated. For applications containing only absolute sections, nothing will be allocated.
- Specified application entry point
- Definition of the reset vector

The minimal linker parameter file will look like this:

```
LINK test.abs  /* Name of the executable file generated. */

NAMES
  test.o       /* Name of the object files in the application. */
END

SEGMENTS
/* READ_ONLY memory area. There should be no overlap between this
   memory area and the absolute sections defined in the assembly
   source file. */

  MY_ROM  = READ_ONLY  0x1000 TO 0x1FFF;
/* READ_WRITE memory area. There should be no overlap between this
   memory area and the absolute sections defined in the assembly
   source file. */

  MY_RAM  = READ_WRITE 0x2000 TO 0x2FFF;
END

PLACEMENT
```

```
/*Relocatable variable sections are allocated in MY_RAM.*/

  .data    INTO MY_RAM;
/*Relocatable code and constant sections are allocated in MY_ROM */

  .text    INTO MY_ROM;
END

INIT entry              /*  Application entry point */
VECTOR ADDRESS 0xFFFE entry /* initialization of the reset vector */
```

NOTE: *Allow no overlap between the absolute section defined in the assembly source file and the memory area defined in the PRM file.*

Since the memory areas (segments) specified in the PRM file are used only to allocate relocatable sections, nothing will be allocated there when the application contains only absolute sections. In that case, specify invalid address ranges in the PRM file.

The module *fiboorg.asm* located in the demo directory is a small example of using absolute sections in an application.

11.4 Working with Relocatable Sections

A relocatable section is a section whereby the start address is determined at link time. (See modules *fibo.asm* and *fibo.prm* in the demo directory.)

11.4.1 Defining Relocatable Sections in the Assembly Source File

A relocatable section is defined using the directive `SECTION`.

Example:

Defining a relocatable section containing data:

```
constSec: SECTION ; Relocatable constant data section
cst1: DC.B    $A6
cst2: DC.B    $BC

dataSec:  SECTION ; Relocatable data section
var:  DS.B    1
```

In the previous portion of code, the label `cst1` will be located at offset 0 from the `constSec` section start address, and label `cst2` will be located at offset 1 from the `constSec` section start address.

Defining a relocatable section containing code:

```
codeSec:  SECTION ; Relocatable code section
entry:
   LDA  cst1      ; Load value in cst1
   ADD  cst2      ; Add value in cst2
   STA  var       ; Store in var
   BRA  entry
```

In the previous portion of code, the instruction `LDA` will be located at offset 0 from the `codeSec` start address, and instruction `ADD` at offset 3 from the `codeSec` start address.

To avoid problems during linking or executing an application, an assembly file must:

- Initialize the stack pointer. The instruction `RSP` can be used to initialize the stack pointer to `$00FF`.
- Publish the application entry point using `XDEF.`

11.4.2 Linking an Application Containing Relocatable Sections

Applications containing relocatable sections must be linked. The linker parameter file must contain at least the:

- Name of the absolute file, which is the name of the object file that should be linked
- Specification of a memory area where the sections containing variables must be allocated
- Specification of a memory area where the sections containing code or constants must be allocated
- Specification of the application entry point
- Definition of the reset vector

The minimal linker parameter file will look like this:

```
LINK test.abs  /* Name of executable file generated */
NAMES
  test.o       /* Name of object files in the application */
END
SEGMENTS
/* READ_ONLY memory area.  */
  MY_ROM  = READ_ONLY  0x0B00 TO 0x0BFF;
/* READ_WRITE memory area. */
  MY_RAM  = READ_WRITE 0x0800 TO 0x08FF;
END
PLACEMENT
/* Relocatable variable sections are allocated in MY_RAM. */
  .data    INTO MY_RAM;
/* Relocatable code and constant sections are allocated in MY_ROM */
  .text    INTO MY_ROM;
END
INIT entry              /*  Application entry point */
VECTOR ADDRESS 0xFFFE entry /* initialization of the reset vector */
```

NOTE: *The programmer should ensure that the memory ranges specified in the SEGMENT block are valid addresses for the MCU being used.*

The module *fibo.asm* located in the demo directory is a small example of using the relocatable sections in an application.

11.5 Initializing the Vector Table

The vector table is initialized in the assembly source file or in the linker parameter file (PRM).

The interrupt and reset vectors location in memory are bound to the specific processor. In this section, the M68HC05C9 is used for all examples. The reset vector for this derivative is located at address $3FFE–$3FFF.

These sources can generate interrupts:

- Software interrupt (SWI), interrupt vector location `$3FFC-$3FFD`
- External interrupt, interrupt vector location `$3FFA-$3FFB`
- Capture/compare timer interrupt, interrupt vector location `$3FF8-$3FF9`
- Serial communication interface interrupt (SCI), interrupt vector location `$3FF6-$3FF7`
- Serial peripheral interface interrupt (SPI), interrupt vector location `$3FF4-$3FF5`

11.5.1 Initializing the Vector Table in the Linker PRM File

Initializing the vector table from the PRM file allows initialization of single entries in the table. All or some of the entries can be initialized in the vector table.

The labels or functions inserted in the vector table must also 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 XIRQFunc, SWIFunc, ResetFunc
DataSec: SECTION SHORT
Data:    DS.W 5    ; Each interrupt increments an element of the table
CodeSec: SECTION

                     ; Implementation of the interrupt functions.
XIRQFunc:
          LDA  #0
          BRA  int
SWIFunc:
          LDA  #4
          BRA  int
ResetFunc:
          LDA  #8
          BRA  entry
int:
          LDX  #Data  ; Load address of symbol Data in X
          ; X <- address of the appropriate element in table
Ofset:    TSTA
          BEQ  Ofset3
Ofset2:
          INCX
          DECA
          BNE  Ofset2
Ofset3:
          INC  0, X   ; table element is incremented
          RTI
entry:
          RSP
          CLRX

          CLI          ; enables interrupts
```

```
loop:     BRA loop
```

NOTE: *The functions `XIRQFunc`, `SWIFunc`, `ResetFunc` are published. This is required because they are referenced in the linker PRM file. All interrupt functions must be terminated with an RTI (return-to-interrupt) instruction.*

The vector table is initialized by the linker command VECTOR ADDRESS.

Example:

```
LINK test.abs
NAMES
  test.o
END
SEGMENTS
  MY_ROM   = READ_ONLY  0x0800 TO 0x08FF;
  MY_RAM   = READ_WRITE 0x0B00 TO 0x0CFF;
  Z_RAM    = READ_WRITE 0x0080 TO 0x00D0;
  MY_STACK = READ_WRITE 0x0D00 TO 0x0DFF;
END
PLACEMENT
  .data        INTO Z_RAM;
  .text        INTO MY_ROM;
  .stack       INTO MY_STACK;
END
INIT ResetFunc
VECTOR ADDRESS 0x3FFA XIRQFunc
VECTOR ADDRESS 0x3FFC SWIFunc
VECTOR ADDRESS 0x3FFE ResetFunc
```

NOTE: *The statement `INIT ResetFunc` defines the application entry point. Usually, this entry point is initialized with the same address as the reset vector. The statement `VECTOR ADDRESS 0x3FFA XIRQFunc` specifies that the address of function `XIRQFunc` should be written at address `0x3FFA`.*

11.5.2 Initializing Vector Table in Assembly Source Files Using a Relocatable Section

Initializing the vector table in the assembly source file requires that all entries in the table are initialized. Interrupts that 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 or an external reference must be available for them. The vector table can be defined in an assembly source file in an additional section containing constant variables.

Example:

```
         XDEF ResetFunc
DataSec: SECTION SHORT
Data:    DS.W 5          ;Each interrupt increments an element of the table.
CodeSec: SECTION
                         ;Implementation of the interrupt functions.
XIRQFunc:
          LDA   #0
          BRA   int
SWIFunc:
          LDA   #4
          BRA   int
ResetFunc:
          LDA   #8
          BRA   entry
SCIFunc:
          LDA   #16
          BRA   entry
DummyFunc:
          RTI
int:
          LDX   #Data     ; Load address of symbol Data in X
                          ; X <- address of appropriate element in table
Ofset:    TSTA
          BEQ   Ofset3
```

```
Ofset2:
          INCX
          DECA
          BNE  Ofset2
Ofset3:
          INC  0, X      ; table element is incremented
          RTI
entry:
          RSP
          CLRX

          CLI            ; enables interrupts

loop:     BRA loop

VectorTable:   SECTION
; Definition of the vector table.
SCIInt:        DC.W SCIFunc
RTIInt:        DC.W DummyFunc
XIRQInt:       DC.W XIRQFunc
SWIInt:        DC.W SWIFunc
ResetInt:      DC.W ResetFunc
```

NOTE: *Each constant in the section `VectorTable` is defined as a word (2-byte constant) because entries in the HC05 vector table are 16 bits wide. In the previous example, the constant `XIRQInt` is initialized with the address of the label `XIRQFunc`. The constant `RTIInt` is initialized with the address of the label `DummyFunc` because this interrupt is not in use. All the labels specified as initialization values must be defined, published (using XDEF), or imported (using XREF), before the vector table section.*

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
SEGMENTS
  MY_ROM   = READ_ONLY  0x0800 TO 0x08FF;
  MY_RAM   = READ_WRITE 0x0B00 TO 0x0CFF;
  Z_RAM    = READ_WRITE 0x0080 TO 0x00D0;
  MY_STACK = READ_WRITE 0x0D00 TO 0x0DFF;
/* Define the memory range for the vector table */
  Vector  = READ_ONLY  0x3FF6 TO 0x3FFF;
END
PLACEMENT
  .data        INTO  Z_RAM;
  .text        INTO MY_ROM;
  .stack       INTO MY_STACK;
/*  Place  the  section  'VectorTable'  at  the
appropriate address */
  VectorTable  INTO Vector;
END
INIT ResetFunc
ENTRIES
  *
END
```

NOTE: *The statement `Vector = READ_ONLY 0x3FF6 TO 0x3FFF` defines the memory range for the vector table. The statement `VectorTable INTO Vector` specifies that the vector table should be loaded in the read-only memory area vector. This means the constant `SCIInt` will be allocated at address 0x3FF6, the constant `RTIInt` will be allocated at address 0x3FF8, the constant `XIRQInt` will be allocated at address 0x3FFA, the constant `SWIInt` will be allocated at address 0x3FFC, and the constant `ResetInt` will be allocated at address 0x3FFE. The statement `ENTRIES * END` switches smart linking OFF. 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.*

11.5.3 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 entries in the table are initialized. Interrupts that are not used must be associated with a standard handler.

The labels or functions that are inserted in the vector table must be implemented in the assembly source file or an external reference must be available for them. The vector table can be defined in an assembly source file in an additional section containing constant variables.

Example:

```
         XDEF ResetFunc
DataSec: SECTION SHORT
Data:    DS.W 5    ;Each interrupt increments an element of the table
CodeSec: SECTION
                   ;Implementation of the interrupt functions
XIRQFunc:
          LDA  #0
          BRA  int
SWIFunc:
          LDA  #4
          BRA  int
ResetFunc:
          LDA  #8
          BRA  entry
SCIFunc:
          LDA  #16
          BRA  entry
DummyFunc:
          RTI
int:
          LDX  #Data  ;Load address of symbol Data in X
                      ;X <- address of the appropriate element in table
Ofset:    TSTA
          BEQ  Ofset3
Ofset2:
          INCX
          DECA
          BNE  Ofset2
```

```
Ofset3:
          INC  0, X    ;table element is incremented
          RTI
entry:
          RSP
          CLRX
          CLI          ;enables interrupts

loop:     BRA loop

               ORG $3FF6
                      ;Definition of vector table in absolute section
                      ; starting at address $3FF6.
SCIInt:        DC.W SCIFunc
RTIInt:        DC.W DummyFunc
XIRQInt:       DC.W XIRQFunc
SWIInt:        DC.W SWIFunc
ResetInt:      DC.W ResetFunc
```

NOTE: *Each constant in the section `VectorTable` is defined as a word (2-byte constant) because the entry in the HC05 vector table is 16 bits wide. In the previous example, the constant `SCIInt` is initialized with the address of the label `SCIFunc`, the constant `RTIInt` is initialized with the address of the label `DummyFunc`, etc. Labels specified as initialization values must be defined, published (using XDEF), or imported (using XREF) before the vector table section. The statement `ORG $3FF6` specifies that the following section must start at address $3FF6.*

The section should be placed at the expected address. This is done in the linker parameter file.

Example:

```
LINK test.abs
NAMES
  test.o
END
SEGMENTS
  MY_ROM   = READ_ONLY  0x0800 TO 0x08FF;
  MY_RAM   = READ_WRITE 0x0B00 TO 0x0CFF;
  Z_RAM    = READ_WRITE 0x0080 TO 0x00D0;
  MY_STACK = READ_WRITE 0x0D00 TO 0x0DFF;
END
PLACEMENT
  .data        INTO  Z_RAM;
  .text        INTO MY_ROM;
  .stack       INTO MY_STACK;
END
INIT ResetFunc
ENTRIES
  *
END
```

NOTE: *The statement `ENTRY  *  END` switches smart linking OFF. 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 objects referenced in the absolute file.*

11.6 Splitting an Application into Different Modules

A complex application or application involving several programmers can be split into several simple modules. To avoid problems when merging modules, for each assembly source file one include file must be created containing the definition of the symbols exported from this module. For symbols referring to code label, a small description of the interface is required.

An example of an assembly file (*Test1.asm*) is:

```
          XDEF AddSource
          XDEF Source

DataSec:  SECTION
Source:   DS.W 1
CodeSec:  SECTION
AddSource:
          RSP
          ADD Source
          STA Source
          RTS
```

an example of a corresponding include file(*Test1.inc*) is:

```
        XREF AddSource
; The function AddSource adds the value stored in the variable
; Source to the content of register A. The result of the computation
; is stored in the variable Source.
; Input Parameter : register A contains the value, which should be
; added to the variable Source.
; Output Parameter: register A contains the result of the addition.
         XREF Source
; The variable Source is a word variable.
```

Each assembly module using a symbol defined in another assembly file should include the corresponding include file.

An example of an assembly file (*Test2.asm*) is:

```
                  XDEF entry
                  INCLUDE “Test1.inc”

          CodeSec:   SECTION
          entry:     RSP
                     LDA #$7
                     JSR AddSource
                    BRA entry
```

The application PRM file must list both object files used to build the application. When a section is present in the different object files, the object file sections are concatenated in a single absolute file section. The different object file sections are concatenated in the order the object files are specified in the PRM file.

An example of a PRM file (*Test2.prm*) is:

```
LINK test2.abs    /* Name of executable file generated */

NAMES
  test1.o test2.o /*Name of object files building the application*/
END
 SEGMENTS
MY_ROM = READ_ONLY  0x0B00 TO 0x0BFF; /* READ_ONLY memory area  */
MY_RAM = READ_WRITE 0x0800 TO 0x08FF; /* READ_WRITE memory area */
END
 PLACEMENT
DataSec,.data INTO MY_RAM; /* variables are allocated in MY_RAM */
CodeSec,.text INTO MY_ROM; /* code and constants */
                           /* are allocated in MY_ROM */
END
INIT entry    /* Definition of the application entry point */
VECTOR ADDRESS 0xFFFE entry /* Definition of the reset vector */
```

NOTE: *The statement NAMES `test1.o test2.o` END lists the two object files building the application. A space character separates the object filenames. The section `CodeSec` is defined in both object files. In `test1.o`, the section `CodeSec` contains the symbol `AddSource`. In `test2.o`, `CodeSec` contains the symbol `entry`. According to the order in which the object files are listed in the NAMES block, the function `AddSource` will be allocated first (at address 0xB00) and symbol `entry` will be allocated next to it.*

11.7 Using Direct Addressing Mode to Access Symbols

The various ways to inform the assembler to use the direct addressing mode on a symbol are discussed here.

11.7.1 Using Direct Addressing Mode to Access External Symbols

External symbols that should be accessed using the direct addressing mode must be declared using the directive `XREF.B` in place of `XREF`.

Example:

```
         XREF.B ExternalDirLabel
         XREF   ExternalExtLabel
…
         LDA   ExternalDirLabel ; Direct addressing mode is used
…
         LDA   ExternalExtLabel ; Extended addressing mode is used
```

11.7.2 Using Direct Addressing Mode to Access Exported Symbols

Symbols that are exported using the directive `XDEF.B`, will be accessed using the direct addressing mode. Symbols that are exported using the directive `XDEF` are accessed using the extended addressing mode.

Example:

```
         XDEF.B DirLabel
         XDEF   ExtLabel
…
         LDA   DirLabel ; Direct addressing mode is used
…
         LDA   ExtLabel ; Extended addressing mode is used
```

11.7.3 Defining Symbols in the Direct Page

Symbols defined in the predefined section BSCT are always accessed using the direct addressing mode.

Example:

```
…
     BSCT
DirLabel: DS.B 3
dataSec: SECTION
ExtLabel: DS.B 5
…
codeSec: SECTION
…
     LDA   DirLabel ; Direct addressing mode is used
…
     LDA   ExtLabel ; Extended addressing mode is used
```

11.7.4 Using a Force Operator

A force operator can be specified in an assembly instruction to force direct or extended addressing mode.

Supported force operators are:

- `<` or `.B` to force direct addressing mode
- `>` or `.W` to force extended addressing mode

Example:

```
…
dataSec: SECTION
label: DS.B 5
…
codeSec: SECTION
…
     LDA   <label ; Direct addressing mode is used
     LDA   label.B; Direct addressing mode is used
…
     LDA   >label ; Extended addressing mode is used
     LDA   label.W ; Extended addressing mode is used
```

11.7.5 Using SHORT Sections

Symbols defined in a section with the qualifier `SHORT` are always accessed using the direct addressing mode.

Example:

```
…
shortSec: SECTION SHORT
DirLabel: DS.B 3
…
dataSec:  SECTION
ExtLabel: DS.B 5
…
codeSec: SECTION
…
     LDA   DirLabel ; Direct addressing mode is used
…
     LDA   ExtLabel ; Extended addressing mode is used
```

11.8 Generating an .abs File Directly

The assembler generates an *.abs* file directly from an assembly source file. A Motorola S file is generated at the same time and can be directly burnt into an EPROM.

11.8.1 Assembler Source File

When an *.abs* file is generated using the assembler (no linker), the application must be implemented in a single assembly unit and contain only absolute sections.

Example:

```
        ABSENTRY entry ; Specifies the application Entry point
        ORG $FFFE     ; Reset vector definition
Reset:  DC.W entry
        ORG $40       ; Define an absolute constant section
var1:   DC.B 5        ; Assign 5 to the symbol var1
        ORG $80       ; Define an absolute data section
data:   DS.B 1        ; Define one byte variable in RAM
address 40
        ORG $B00      ; Define an absolute code section
entry:
        RSP           ; Load stack pointer
        LDA   var1
main:
        INCA
        STA   data
       BRA   main
```

When writing an assembly source file for direct absolute file generation, pay special attention to these points:

- The directive `ABSENTRY` is used to write the entry point address in the generated absolute file. To set the entry point of the application on the label `entry` in the absolute file, this code is needed: `ABSENTRY entry`.
- The reset vector must be initialized in the assembly source file, specifying the application entry point. An absolute section is created at the reset vector address. This section contains the application entry point address.

  To set the entry point of the application at address $FFFE on the label `entry`, this code is needed:

  ```
  ORG $FFFE  ; Reset vector definition
             ; Reset: DC.W entry
  ```

- It is strongly recommended to use separate sections for code, data, and constants. All sections used in the assembler application must be absolute. They must be defined using the `ORG` directive. The address for constant or code sections has to be located in the ROM memory area, while the data sections have to be located in RAM (according to the hardware being used). Also, ensure that no sections overlap.

11.8.2 Assembling and Generating the Application

Once the source file is available, it can be assembled.

1. Start the assembler by clicking the **ezASM** icon in the MCUez toolbar. Enter the name of the file to be assembled in the editable combo box, for example *abstest.asm*.

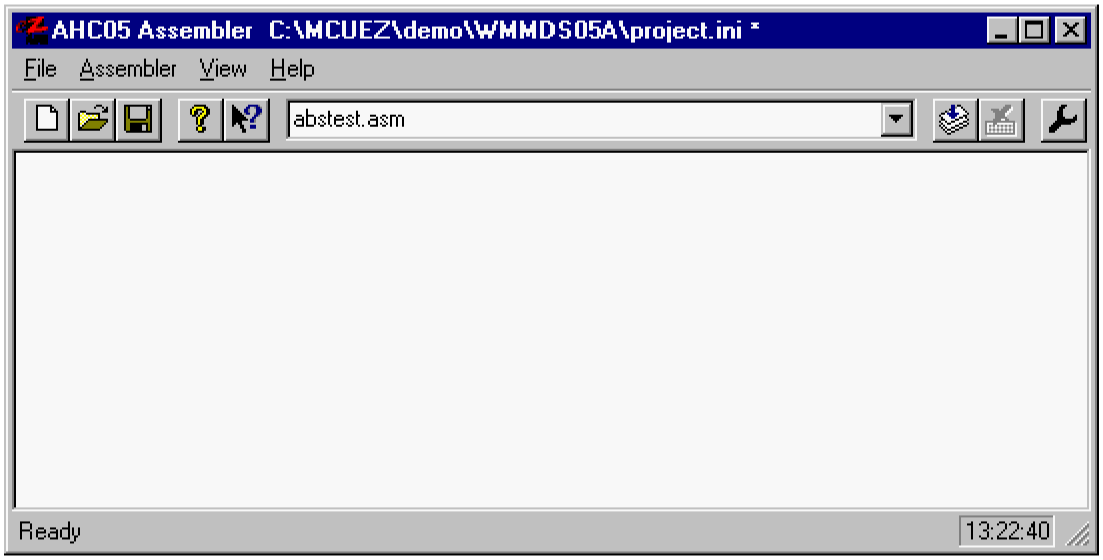
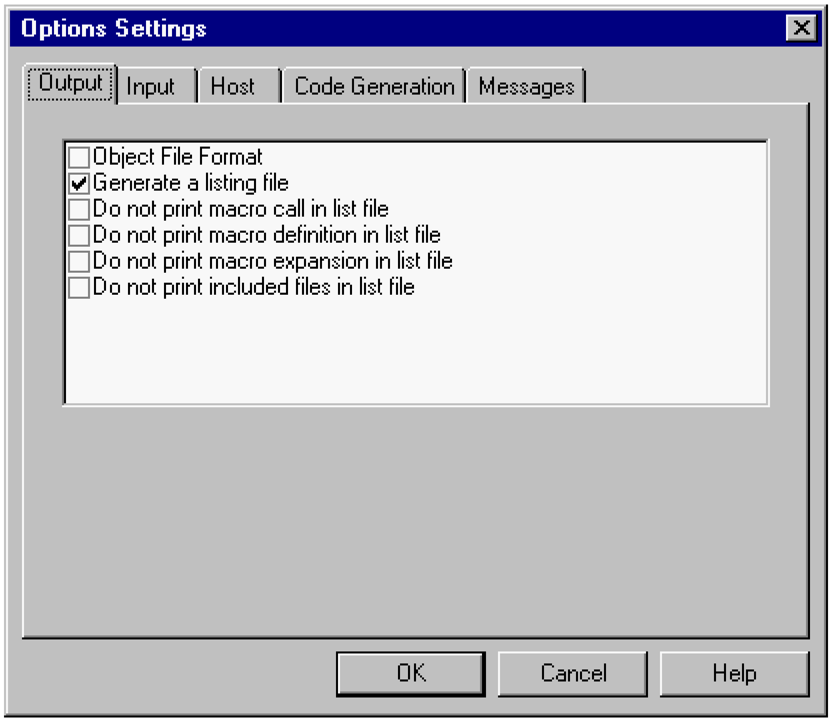


Figure 11-1. Starting the MCUez Assembler

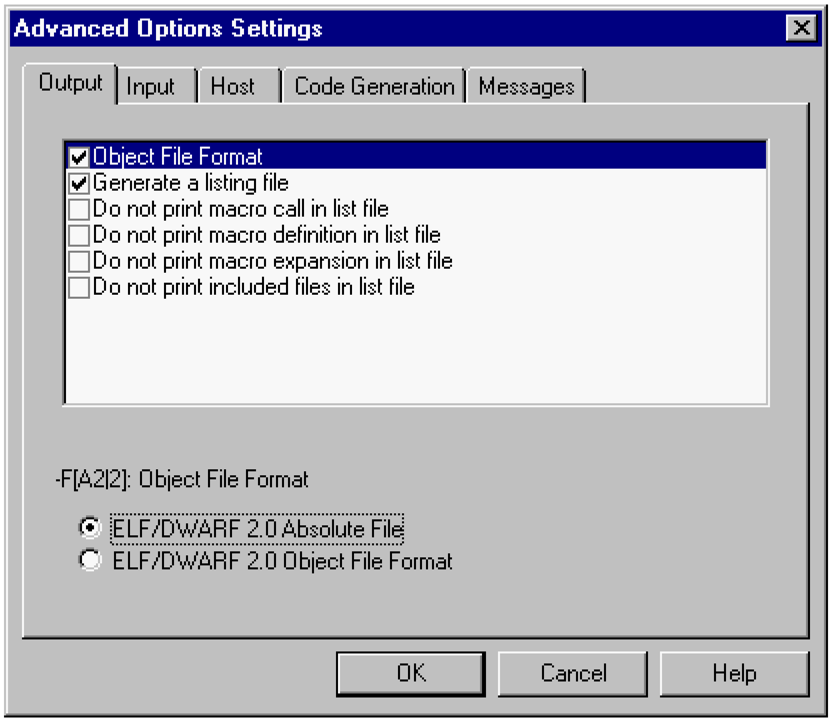
2. Select the menu entry **Assembler | Options** to display the **Options Settings** dialog box.



☐ Object File Format
☑ Generate a listing file
☐ Do not print macro call in list file
☐ Do not print macro definition in list file
☐ Do not print macro expansion in list file
☐ Do not print included files in list file

Figure 11-2. Options Setting Dialog Box

3. In the **Output** folder, select the check box in front of the label **Object File Format**.



Advanced Options Settings

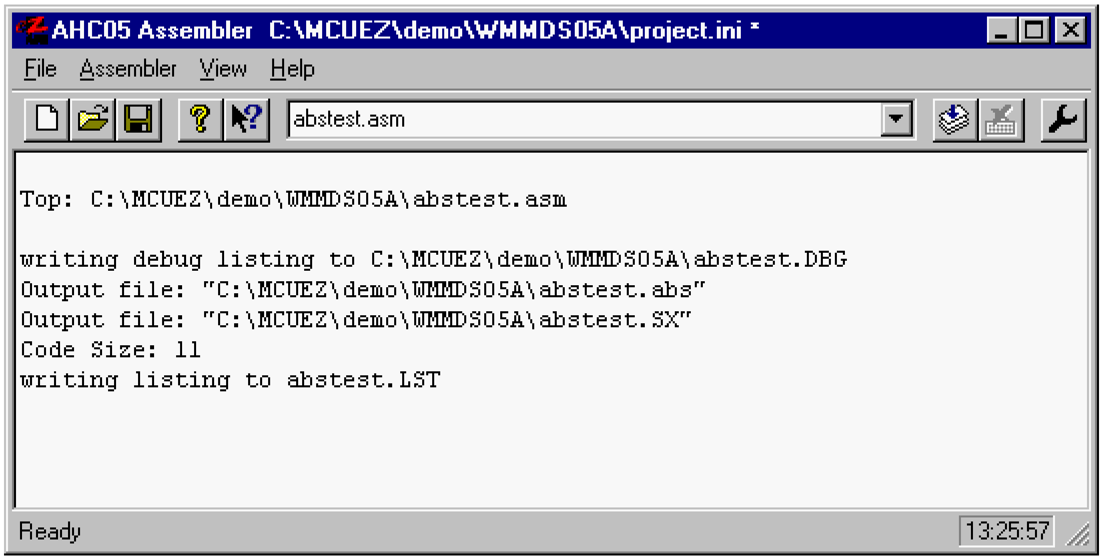
- [✓] Object File Format
- [✓] Generate a listing file
- [ ] Do not print macro call in list file
- [ ] Do not print macro definition in list file
- [ ] Do not print macro expansion in list file
- [ ] Do not print included files in list file

-F[A2|2]: Object File Format

[✓ ELF/DWARF 2.0 Absolute File]
ELF/DWARF 2.0 Object File Format

Figure 11-3. Selecting the Object File Format

4. Select the radio button **ELF/DWARF 2.0 Absolute File** and click **OK** to generate an absolute file. Click on the **Assemble** button to assemble the file. The assembly process is shown in **Figure 11-4**.



```
Top: C:\MCUEZ\demo\WMMDS05A\abstest.asm

writing debug listing to C:\MCUEZ\demo\WMMDS05A\abstest.DBG
Output file: "C:\MCUEZ\demo\WMMDS05A\abstest.abs"
Output file: "C:\MCUEZ\demo\WMMDS05A\abstest.SX"
Code Size: 11
writing listing to abstest.LST
```

Figure 11-4. Generating an Assembler .abs File

The generated absolute file (*.abs*) is used with the target board or emulator. The file can be downloaded directly to the HC05 target. The target must be reset. Select menu entry **MMDS0508 | Reset** before running the application. The *.sx* file that is generated is a standard Motorola S record file. This file can be directly burnt into an EPROM.

Section 12. Assembler Messages

12.1 Contents

12.2 Introduction. 237
12.2.1 Warning 237
12.2.2 Error 237
12.2.3 Fatal 237

12.3 Message Codes 237
12.3.1 A1000: Conditional Directive Not Closed. 238
12.3.2 A1001: Conditional Else Not Allowed Here 239
12.3.3 A1051: Zero Division in Expression 240
12.3.4 A1052: Right Parenthesis Expected. 241
12.3.5 A1053: Left Parenthesis Expected. 242
12.3.6 A1101: Illegal Label: Label Is Reserved 243
12.3.7 A1103: Illegal Definition of Label. 244
12.3.8 A1104: Undeclared User-Defined Symbol <symbolName> 245
12.3.9 A2301: Label Is Missing 246
12.3.10 A2302: Macro Name Is Missing 246
12.3.11 A2303: ENDM Is Illegal 247
12.3.12 A2304: Macro Definition Within Definition 248
12.3.13 A2305: Illegal Redefinition of Instruction or Directive Name . . 249
12.3.14 A2306: Macro Not Closed at End of Source 250
12.3.15 A2307: Macro Redefinition 251
12.3.16 A2308: Filename Expected 252
12.3.17 A2309: File Not Found 252
12.3.18 A2310: Illegal Size Char 253
12.3.19 A2311: Symbol Name Expected 254
12.3.20 A2312: String Expected. 255
12.3.21 A2313: Nesting of Include Files Exceeds 50. 256
12.3.22 A2314: Expression Must be Absolute 256
12.3.23 A2316: Section Name Required 257
12.3.24 A2317: Illegal Redefinition of Section Name 258
12.3.25 A2318: Section Not Declared 259
12.3.26 A2320: Value Too Small 260

12.3.27 A2321: Value Too Big 261
12.3.28 A2323: Label Is Ignored 262
12.3.29 A2324: Illegal Base (2, 8, 10, and 16) 263
12.3.30 A2325: Comma or Line End Expected 264
12.3.31 A2326: Label is Redefined 265
12.3.32 A2327: ON or OFF Expected 266
12.3.33 A2328: Value Is Truncated 266
12.3.34 A2329: FAIL Found. 267
12.3.35 A2330: String is Not Allowed 268
12.3.36 A2332: FAIL Found. 269
12.3.37 A2333: Forward Reference Not Allowed 270
12.3.38 A2334: Only Labels Defined in the Current Assembly
Unit Can Be Referenced in an EQU Expression 271
12.3.39 A2335: Exported Absolute EQU Label Is Not Supported. 272
12.3.40 A2336: Value Too Big 273
12.3.41 A2338: <Message String> 274
12.3.42 A2341: Relocatable Section Not Allowed: an Absolute
File Is Generated 275
12.3.43 A13001: Illegal Addressing Mode 276
12.3.44 A13005: Comma Expected 277
12.3.45 A13007: Relative Branch with Illegal Target 278
12.3.46 A13008: Illegal Expression 280
12.3.47 A13101: Illegal Operand Format 280
12.3.48 A13102: Operand Not Allowed 281
12.3.49 A13106: Illegal Size Specification for HC05 Instruction 281
12.3.50 A13108: Illegal Character at the End of Line 282
12.3.51 A13109: Positive Value Expected 283
12.3.52 A13110: Mask Expected 284
12.3.53 A13111: Value Out of Range. 285
12.3.54 A13201: Lexical Error in First or Second Field 286
12.3.55 A13202: Not an HC05 Instruction or Directive. 287
12.3.56 A13401: Value Out of Range — 128...127 288
12.3.57 A13403: Complex Relocatable Expression Not Supported. 289
12.3.58 A13405: Code Size Per Section is Limited to 32 Kbytes 290
12.3.59 A13406: HIGH with Initialized RAM Not Supported. 291
12.3.60 A13407: LOW with Initialized RAM Not Supported 291
12.3.61 A13601: Error in Expression 291
12.3.62 A13602: Error at End of Expression 292

12.2 Introduction

The assembler can generate three types of messages:

1. Warning
2. Error
3. Fatal

12.2.1 Warning

A message will be printed and assembly will continue. Warning messages are used to indicate possible programming errors to the user.

12.2.2 Error

A message will be printed and assembly will stop. Error messages are used to indicate illegal language usage.

12.2.3 Fatal

A message will be printed and assembly will be aborted. A fatal message indicates a severe error that will stop the assembling.

12.3 Message Codes

If the assembler prints out a message, the message contains a message code (A for assembler) and a 4- to 5-digit number. This number may be used to search for the indicated message in the manual. All messages generated by the assembler are documented in increasing order for easy and fast 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.

12.3.1 A1000: Conditional Directive not Closed

Type: Error

Description: A conditional block is not closed. A conditional block can be opened by one of these directives:

IF, IFEQ, IFNE, IFLT, IFLE, IFGT, IFGE, IFC, IFNC, IFDEF, IFNDEF

Example:

```
        IFEQ (defineConst)
const1:   DC.B 1
const2:   DC.B 2
```

Tip: Close the conditional block with an ENDIF or ENDC directive.

Example:

```
        IFEQ (defineConst)
const1:   DC.B 1
const2:   DC.B 2
        ENDIF
```

Be careful: A conditional block that starts inside a macro must be closed within the same macro.

Example: The following portion of code generates an error because the conditional block `IFEQ` is opened within the macro `MyMacro` and closed outside the macro.

```
MyMacro: MACRO
        IFEQ (SaveRegs)
          NOP
          NOP
          ENDM
          NOP
        ENDIF
```

12.3.2 A1001: Conditional Else not Allowed Here

Type:	Error
Description:	A second ELSE directive is detected in a conditional block.

Example:

```
  IFEQ (defineConst)
...
  ELSE
...
  ELSE
...
  ENDIF
```

Tip: Remove the extra ELSE directive.

Example:

```
  IFEQ (defineConst)
...
  ELSE
...
  ENDIF
```

12.3.3 A1051: Zero Division in Expression

Type: Error

Description: A zero division is detected in an expression.

Example:

```
label:  EQU 0;
label2: EQU $5000
...
        LDX #(label2/label)
```

Tip: Modify the expression or specify it in a conditional assembly block.

Example:

```
label:  EQU 0;
label2: EQU $5000
...
     IFNE (label)
        LDX #(label2/label)
     ELSE
        LDX #label2
     ENDIF
```

12.3.4 A1052: Right Parenthesis Expected

Type: Error

Description: A right parenthesis is missing in an assembly expression or expression containing a HIGH or LOW operator.

Example:

```
MyData: SECTION
variable: DS.B 1

label:  EQU (2*4+6
label2: EQU PAGE (variable
label5: EQU HIGH (variable
```

Tip: Insert the right parenthesis at the correct position.

Example:

```
MyData: SECTION
variable: DS.B 1

label:  EQU (2*4)+6
label2: EQU PAGE(variable)
label5: EQU HIGH (variable)
```

12.3.5 A1053: Left Parenthesis Expected

Type: Error

Description: A left parenthesis is missing in an expression containing a reference to the page (bank) where an object is allocated.

Example:

```
MyData: SECTION
variable: DS.B 1
label3: EQU PAGE variable)
label5: EQU HIGH variable)
```

Tip: Insert the left parenthesis at the correct position.

Example:

```
MyData: SECTION
variable: DS.B 1

label3: EQU PAGE (variable)
label5: EQU HIGH (variable)
```

12.3.6 A1054 References on Non-Absolute Objects not Allowed When Options -FA1 or -FA2 Are Enabled

Type:	Error
Description	A reference to a relocatable object has been detected during generation of an absolute file by the assembler.

Example:

```
XREF extData
DataSec: SECTION
data1:   DS.W 1
         ORG $800
entry:
         LDX #data1
         LDX extData
```

Tips:

•When generating an absolute file, the application should be encoded in a single source file, and should only contain a relocatable symbol.

•To avoid this message, define sections as absolute sections and remove all XREF directives from the source file.

Example:

```
         ORG $B00
data1:   DS.W 1
         ORG $800
entry:
         LDX #data1
```

12.3.7 A1101: Illegal Label: Label is Reserved

Type: Error

Description: A reserved identifier is used as a label. Reserved identifiers are:

- Mnemonics associated with target processor registers are A, CCR, SP, X .
- Mnemonics associated with special target processor operator are HIGH and LOW.

Example:

```
A:     NOP
       NOP
       RTS
```

Tip: Modify the name of the label to an identifier that is not reserved.

Example:

```
ASub:  NOP
       NOP
       RTS
```

12.3.8 A1103: Illegal Redefinition of Label

Type: Error

Description: The label specified in front of a comment, assembly instruction, or directive is detected twice in a source file.

Example:

```
            XDEF Entry

DataSec1:   SECTION
label1:     DS.W 2
label2:     DS.L 2

CodeSec1:   SECTION
MySub:      LDX #label1
            CPX #$500
            BNE label2
            NOP
            NOP
            NOP
label2:     RTS

Entry:      RSP

main:       BSR MySub
            BRA main
```

Tip: Rename one of the label names.

Example:

```
            XDEF Entry

DataSec1:   SECTION
DataLab1:   DS.W 2
DataLab2:   DS.L 2

CodeSec1:   SECTION
MySub:      LDX #DataLab1
            CPX #$500
            BNE CodLab2
            NOP
            NOP
            NOP
CodLab2:    RTS

Entry:      RSP

main:       BSR MySub
            BRA main
```

12.3.9 A1104: Undeclared User-Defined Symbol <symbolName>

Type:	Error
Description:	The label <symbolName> is referenced in the assembly file, but it is never defined.

Example:

```
Entry:
   LDA #56
   STA Variable
   RTS
```

Tip: The label <symbolName> must be defined either in the current assembly file or specified as an external label.

Example:

```
   XREF Variable
...
...
Entry:
   LDA #56
   STA Variable
   RTS
```

12.3.10 A2301: Label is Missing

Type: Error

Description: A label name is missing at the front of an assembly directive requiring a label. These directives are SECTION, EQU, and SET.

Example:

```
   SECTION 4
   …
   EQU $67
   …
   SET $77
```

Tip: Insert a label in front of the directive.

Example:

```
codeSec: SECTION 4
   …
myConst: EQU $67
   …
mySetV:  SET $77
```

12.3.11 A2302: Macro Name is Missing

Type: Error

Description: A label name is missing in front of a MACRO directive.

Example:

```
   MACRO
     LDA \1
     ADD \2
     STA \1
   ENDM
```

Tip: Insert a label in front of the MACRO directive.

Example:

```
AddM: MACRO
        LDA \1
        ADD \2
        STA \1
      ENDM
```

12.3.12 A2303: ENDM is Illegal

Type: Error

Description: An ENDM directive is detected outside a macro.

Example:

```
AddM: MACRO
        LDA \1
        ADD \2
        STA \1
      ENDM
      NOP
      AddM data1, data2
      ENDM
```

Tip: Remove the superfluous ENDM directive.

Example:

```
AddM: MACRO
        LDA \1
        ADD \2
        STA \1
      ENDM
      NOP
      AddM data1, data2
```

12.3.13 A2304: Macro Definition Within Definition

Type:	Error
Description:	A macro definition is detected inside another macro definition. The assembler does not support this.

Example:

```
AddM: MACRO
AddX: MACRO
        LDX \1
        INCX
        STX \1
      ENDM
        LDA \1
        ADD \2
        STA \1
      ENDM
```

Tip: Define the second macro outside the first one.

Example:

```
AddX: MACRO
        LDX \1
        INCX
        STX \1
      ENDM
AddM: MACRO
        LDA \1
        ADD \2
        STA \1
      ENDM
```

12.3.14 A2305: Illegal Redefinition of Instruction or Directive Name

Type: Error

Description: An assembly directive or an HC05 instruction name has been used as a macro name. This is not allowed. The assembler cannot determine if the symbol refers to the macro or the instruction.

Example:

```
ADDD: MACRO
        LDA \1
        ADD \2
        STA \1
      ENDM
```

Tip:

Change the name of the macro to an unused identifier.

Example:

```
ADDM: MACRO
        LDA \1
        ADD \2
        STA \1
      ENDM
```

12.3.15 A2306: Macro not Closed at End of Source

Type: Error

Description: An ENDM directive is missing at the end of a macro. The end of the input file is detected before the end of the macro.

Example:

```
AddM: MACRO
       LDA \1
       ADD \2
       STA \1
  NOP
  AddM data1, data2
```

Tip: Insert the missing ENDM directive at the end of the macro.

Example:

```
AddM: MACRO
       LDA \1
       ADD \2
       STA \1
     ENDM
  NOP
  AddM: data1, data2
```

12.3.16 A2307: Macro Redefinition

Type: Error

Description: The input file contains definitions for two macros with the same name.

Example:

```
AddM MACRO
       LDX \1
       INCX
       STX \1
     ENDM
…
AddM MACRO
       LDA \1
       ADD \2
       STA \1
     ENDM
```

Tip: Change the name of one of the macros to generate unique identifiers.

Example:

```
AddX MACRO
       LDX \1
       INCX
       STX \1
     ENDM
AddM MACRO
       LDA \1
       ADD \2
       STA \1
     ENDM
```

12.3.17 A2308: Filename Expected

Type: Error

Description: A filename is expected in an INCLUDE directive.

Example:

```
xxx: EQU  $56
     …
     INCLUDE xxx
```

Tip: Specify a filename after the INCLUDE directive.

Example:

```
xxx: EQU  $56
     …
     INCLUDE “xxx.inc”
```

12.3.18 A2309: File not Found

Type: Error

Description: The assembler cannot locate a file specified in the INCLUDE directive.

Tip: If the file exists, check if the directory is specified in the GENPATH environment variable. Ensure that the project directory is correct and a *default.env* file exists. The assembler looks for include files in the project directory and directories listed in the GENPATH environment variable. If the file does not exist, create it or remove the include directive.

12.3.19 A2310: Illegal Size Character

Type: Error

Description: An invalid size specification character is detected in a DCB, DC, DS, FCC, FCB, FDB, RMB, XDEF, or XREF directive.

For XDEF and XREF directives, valid size specification characters are:

- `.B` for symbols located in a section where direct addressing mode can be used
- `.W` for symbols located in a section where extended addressing mode must be used

For DCB, DC, DS, FCC, FCB, FDB, and RMB directives, valid size specification characters are:

- `.B` for byte variables
- `.W` for word variables
- `.L` for long variables

Example:

```
DataSec: SECTION
label1: DS.Q 2
…
ConstSec: SECTION
label2: DC.I 3, 4, 6
```

Tip: Change the size specification character to a valid one.

Example:

```
DataSec: SECTION
label1: DS.L 2
…
ConstSec: SECTION
label2: DC.W 3, 4, 6
```

12.3.20 A2311: Symbol Name Expected

Type: Error

Description: A symbol name is missing after an XDEF, XREF, IFDEF, or IFNDEF directive.

Example:

```
          XDEF $5645
          XREF ; This is a comment
CodeSec:  SECTION
…
          IFDEF $5634
```

Tip: Insert a symbol name at the requested position.

Example:

```
          XDEF exportedSymbol
        XREF importedSymbol; This is a comment
CodeSec:  SECTION
…
          IFDEF changeBank
```

12.3.21 A2312: String Expected

Type: Error

Description A character string is expected at the end of an FCC, IFC, or IFNC directive.

Example:

```
expr:     EQU $5555
expr2:    EQU 5555
DataSec:  SECTION
label:    FCC expr
          …
CodeSec:  SECTION
          …
          IFC expr, expr2
```

Tip: Insert a character string at the requested position.

Example:

```
expr:     EQU $5555
expr2:    EQU 5555
DataSec:  SECTION
label:    FCC “This is a string”
          …
```

12.3.22 A2313: Nesting of Include Files Exceeds 50

Type: Error

Description: The maximum number of nested include files has been exceeded. The assembler supports up to 50 nested include files.

Tip: Reduce the number of nested include files to 50.

12.3.23 A2314: Expression Must be Absolute

Type: Error

Description: An absolute expression is expected at the specified position. Assembler directives expecting an absolute value are: OFFSET, ORG, ALIGN, SET, BASE, DS, LLEN, PLEN, SPC, TABS, IF, IFEQ, IFNE, IFLE, IFLT, IFGE, IFGT.

The first operand in a DCB directive must be absolute:

Example:

```
DataSec: SECTION
label1:  DS.W 1
label2:  DS.W 2
label3:  EQU  8
…
codeSec: SECTION
…
         BASE label1
…
         ALIGN label2
```

Tip: Specify an absolute expression at the specified position.

Example:

```
DataSec: SECTION
label1:  DS.W 1
label2:  DS.W 2
label3:  EQU  8
…
codeSec: SECTION
…
         BASE label3
…
         ALIGN 4
```

12.3.24 A2316: Section Name Required

Type: Error

Description: A SWITCH directive is not followed by a symbol name. Absolute expressions or strings are not allowed in a SWITCH directive.

The symbol specified in a SWITCH directive must refer to a previously defined section.

Example:

```
dataSec: SECTION
label1: DS.B 1
…
codeSec: SECTION
…
         SWITCH $A344
…
```

Tip: Specify the name of a previously defined section in the SWITCH directive.

Example:

```
dataSec: SECTION
label1: DS.B 1
…
codeSec: SECTION
…
         SWITCH dataSec
…
```

12.3.25 A2317: Illegal Redefinition of Section Name

Type: Error

Description: The name associated with a section is previously used as a label in a code or data section or is specified in an XDEF directive.

The macro assembler does not allow a section name to be exported or to use the same name for a section and a label.

Example:

```
dataSec:  SECTION
secLabel: DS.B 1
label1:   DS.B 2
label2:   DS.B 1
...
secLabel: SECTION
          LDA #1
…
```

Tip: Change section name to a unique identifier.

Example:

```
dataSec:  SECTION
data:     DS.B 1
label1:   DS.B 2
label2:   DS.B 1
...
codeSec:  SECTION
          LDA #1
…
```

12.3.26 A2318: Section not Declared

Type:	Error
Description:	The label specified in a SWITCH directive is not associated with a section.

Example:

```
dataSec: SECTION
label1: DS.B 1
…
codeSec: SECTION
…
         SWITCH daatSec
…
```

Tip:	Specify the name of a previously defined section in the SWITCH directive.

Example:

```
dataSec: SECTION
label1: DS.B 1
…
codeSec: SECTION
…
         SWITCH dataSec
…
```

12.3.27 A2320: Value too Small

Type: Error

Description: The absolute expression specified in a directive is too small.

This message can be generated when:

- The expression specified in an ALIGN, DCB, or DS directive is smaller than 1.
- The expression specified in a PLEN directive is smaller than 10. A header is generated at the top of each page of the listing file. The header contains at least six lines, so a page length should be larger than 10 lines to be useful.
- The expression specified in a LLEN, SPC, or TABS directive is smaller than 0 (negative).

Example:

```
         PLEN   5
         LLEN  -4
dataSec: SECTION
         ALIGN  0
…
label1:  DS.W   0
…
```

Tip: Modify the absolute expression to a value in the correct range (see **12.3.28 A2321: Value too Big**).

Example:

```
         PLEN   50
         LLEN   40
dataSec: SECTION
         ALIGN  8
…
label1:  DS.W   1
…
```

12.3.28 A2321: Value too Big

Type: Error

Description: The absolute expression specified in a directive is too big.

This message can be generated if:

- The expression specified in an ALIGN directive is greater than 32,767.
- The expression specified in a DS or DCB directive is greater than 4096.
- The expression specified in a PLEN directive is greater than 10,000.
- The expression specified in an LLEN directive is greater than 132.
- The expression specified in an SPC directive is greater than 65.
- The expression specified in a TABS directive is greater than 128.

Example:

```
          PLEN   50000
          LLEN  200
dataSec:  SECTION
          ALIGN  40000
…
label1:   DS.W   5000
…
```

Tip: Modify the absolute expression to a value in the range specified in the bullet list here.

Example:

```
          PLEN   50
          LLEN   40
dataSec:  SECTION
          ALIGN  8
…
label1:   DS.W   1
…
```

12.3.29 A2323: Label is Ignored

Type: Warning

Description: A label is specified in front of a directive that does not accept a label. The assembler ignores such labels. These labels cannot be referenced anywhere else in the application.

Labels will be ignored in front of these directives:
ELSE, ENDIF, END, ENDM, INCLUDE, CLIST, ALIST, FAIL, LIST, MEXIT, NOLIST, NOL, OFFSET, ORG, NOPAGE, PAGE, LLEN, PLEN, SPC, TABS, TITLE, TTL.

Example:

```
CodeSec: SECTION
         LDA #$5444
label:   PLEN 50
...
label2:  LIST
...
```

Tip: Remove the label that is not required. If a label was needed at that position in a section, define the label on a separate line.

Example:

```
CodeSec: SECTION
         LDA #$5444
label:
         PLEN 50
…
label2:
         LIST
…
```

12.3.30 A2324: Illegal Base (2, 8, 10, and 16)

Type: Error

Description: An invalid base number follows a BASE directive. Valid base numbers are 2, 8, 10, or 16. The expression specified in a BASE directive must be an absolute expression and must match one of the values listed here.

Example:

```
          BASE  67
…
dataSec: SECTION
label:   DS.B 8
…
         BASE  label
```

Tip: Specify a valid value in the BASE directive.

Example:

```
          BASE  16
…
dataSec: SECTION
label:   EQU 8
…
         BASE  label
```

12.3.31 A2325: Comma or Line End Expected

Type: Error

Description: An incorrect syntax has been detected in a DC, FCB, FDB, XDEF, PUBLIC, GLOBAL, XREF, or EXTERNAL directive. This error message is generated when the values listed in one of these directives are not terminated by an end of line character or separated by a comma.

Example:

```
          XDEF  aa1 aa2 aa3 aa4
          XREF  bb1, bb2, bb3, bb4    This is a
comment
…
dataSec: SECTION
dataLab1: DC.B 2 | 4 | 6 | 8
dataLab2: FCB  45, 66, 88     label3:DC.B 4
```

Tip: Use a comma to separate items in the list or insert an end of line character at the end of the list.

Example:

```
          XDEF  aa1, aa2, aa3, aa4
          XREF  bb1, bb2, bb3, bb4    ;This is a
comment
…
dataSec: SECTION
dataLab1: DC.B 2, 4, 6, 8
dataLab2: FCB  45, 66, 88
label3:   DC.B 4
```

12.3.32 A2326: Label is Redefined

Type: Error

Description: A label redefinition has been detected. This message is issued when:

- The label specified in front of a DC, DS, DCB, or FCC directive is already defined.
- One of the label names listed in an XREF directive is already defined.
- The label specified in front of an EQU directive is already defined.
- The label specified in front of a SET directive is already defined and not associated with another SET directive.
- A label with the same name as an externally referenced symbol is defined in the source file.

Example:

```
DataSec: SECTION
label1:  DS.W   4
…
         BSCT
label1: DS.W    1
```

Tip: Modify source code to use unique identifiers.

Example:

```
DataSec: SECTION
data_label1:   DS.W   4
…
         BSCT
bsct_label1: DS.W    1
```

12.3.33 A2327: ON or OFF Expected

Type: Error

Description: The syntax for an MLIST or CLIST directive is not correct. These directives expect a unique operand with a value of ON or OFF.

Example:

```
CodeSec: SECTION
         …
         CLIST
         …
```

Tip: Specify either ON or OFF after the MLIST or CLIST directive.

Example:

```
CodeSec: SECTION
         …
         CLIST ON
         …
```

12.3.34 A2328: Value is Truncated

Type: Warning

Description: The size of one of the constants listed in a DC directive is bigger than the size specified in the DC directive.

Example:

```
DataSec: SECTION
cst1:    DC.B  $56, $784, $FF
cst2:    DC.w  $56, $784, $FF5634
```

Tip: Reduce the value of the constant to the correct value specified in the DC directive.

Example:

```
DataSec: SECTION
cst1:    DC.B  $56, $7, $84, $FF
cst2:    DC.W  $56, $784, $FF, $5634
```

12.3.35 A2329: FAIL Found

Type: Error

Description: The FAIL directive followed by a number smaller than 500 has been detected in the source file. This is normal for the FAIL directive. The FAIL directive is intended for use with conditional assembly to detect a user-defined error or warning condition.

Example:

```
cpChar: MACRO
          IFC "\1", ""
            FAIL 200
            MEXIT
          ELSE
            LDA \1
          ENDIF

          IFC "\2", ""
            FAIL 600
          ELSE
            STA \2
          ENDIF
        ENDM
codeSec: SECTION
Start:
        cpChar , char2
```

12.3.36 A2330: String is not Allowed

Type: Error

Description: A string has been specified as the initial value in a DCB directive. The initial value for a constant block can be any byte, word, or long absolute expression or a simple relocatable expression.

Example:

```
CstSec: SECTION
label:  DCB.B  10, “aaaaa”
…
```

Tip: Specify the ASCII code equivalent for the characters in the string.

Example:

```
CstSec: SECTION
label:  DCB.B  5, $61
```

12.3.37 A2332: FAIL Found

Type:	Warning
Description:	The FAIL directive followed by a number larger than 500 has been detected in the source file. This is normal for the FAIL directive. The FAIL directive is intended for use with conditional assembly to detect a user-defined error or warning condition.

Example:

```
cpChar: MACRO
          IFC "\1", ""
            FAIL 200
            MEXIT
          ELSE
            LDA \1
          ENDIF

          IFC "\2", ""
            FAIL 600
          ELSE
            STA \2
          ENDIF
        ENDM
codeSec: SECTION
Start:
        cpChar char1
```

12.3.38 A2333: Forward Reference not Allowed

Type: Error

Description: A forward reference has been detected in an EQU instruction. This is not allowed.

Example:

```
CstSec: SECTION
label:  DCB.B  10, $61
equLab: EQU label2
...
label2: DC.W $6754
…
```

Tip: Move the EQU instruction after the label definition it refers to.

Example:

```
CstSec: SECTION
label:  DCB.B  10, $61
...
label2: DC.W $6754
…
equLab: EQU label2 + 1
```

12.3.39 A2334: Only Labels Defined in the Current Assembly Unit Can Be Referenced in an EQU Expression

Type:	Error
Description:	One of the symbols specified in an EQU expression is an external symbol, which was previously specified in an XREF directive. This is not allowed due to a limitation in the *ELF* file format.

Example:

```
        XREF label
CstSec: SECTION
lab: DC.B 6
...
equLabel: EQU label+6
...
```

Tip: EQU label containing a reference to an object must be defined in the same assembly module as the object it refers to. Then the EQU label can be exported to other modules in the application.

Example:

```
        XDEF label, equlabel
        ...
CstSec: SECTION
lab:   DC.B 6
label: DC.W 6
...
equLabel: EQU label+6
...
```

12.3.40 A2335: Exported Absolute EQU Label is not Supported

Type: Error

Description: A label specified in front of an EQU directive and initialized with an absolute value was previously specified in an XDEF directive. This is not allowed due to a limitation in the *ELF* file format.

Example:

```
        XDEF equLabel
CstSec: SECTION
lab: DC.B 6
...
equLabel: EQU $77AA
...
```

Tip: EQU labels initialized with an absolute expression can be defined in a special file, which can be included in each assembly file where the labels are referenced.

Example: File *const.inc*

```
...
equLabel: EQU $77AA
...
File Test.asm
        INCLUDE “const.inc”
CstSec: SECTION
lab: DC.B 6
...
```

12.3.41 A2336: Value too Big

Type:	Warning
Description:	The absolute expression specified as an initialization value for a block defined by DCB is too big. This message is generated when the initial value specified in a DCB.B directive cannot be coded on a byte. In this case, the value used to initialize the constant block will be truncated to a byte value.

Example:

```
constSec: SECTION
…
label1:  DCB.B   2, 312
…
```

In the previous example, the constant block is initialized with the value $38 (= 312 and $FF).

Tip: To avoid this warning, modify the initialization value to a byte value.

Example:

```
constSec: SECTION
…
label1:  DCB.B   2, 56
…
```

12.3.42 A2338: <Message String>

Type:	Error
Description:	The FAIL directive followed by a string has been detected in the source file.
	This is normal for the FAIL directive. The FAIL directive is intended for use with conditional assembly to detect a user-defined error or warning condition.

Example:

```
cpChar: MACRO
        IFC "\1", ""
          FAIL "A char must be specified as first parameter"
          MEXIT
        ELSE
          LDA \1
        ENDIF

        IFC "\2", ""
          FAIL 600
        ELSE
          STA \2
        ENDIF
      ENDM
codeSec: SECTION
Start:
            cpChar , char2
```

12.3.43 A2341: Relocatable Section not Allowed: Absolute File is Generated

Type: Error

Description: A relocatable section has been detected while the assembler tries to generate an absolute file. This is not allowed.

Example:

```
DataSec: SECTION
data1:   DS.W 1
         ORG $800
entry:
         LDX #data1
```

Tips:

•When generating an absolute file, the application should be encoded in a single assembly unit and should not contain a relocatable symbol.

•To avoid this message, define all sections as absolute sections and remove all XREF directives from the source file.

Example:

```
         ORG $B00
data1:   DS.W 1
         ORG $800
entry:
         LDX #data1
```

12.3.44 A13001: Illegal Addressing Mode

Type:	Error
Description:	An illegal addressing mode has been detected in an instruction. This can be generated when incorrect code is used for an addressing mode.

Example:

```
LDA X,0
LDA ,X,0
AND 0xFA
```

Tip: Use valid notation for addressing mode encoding.

Example:

```
LDA 0,X
AND #$FA
```

12.3.45 A13005: Comma Expected

Type: Error

Description: A comma (,) is missing between two instructions or directive operands. This error occurs when the comma is missing in a memory block definition using DCB or comparing strings using the directives IFC or IFNC.

Example:

```
MemBlock: DCB.B 8 $00
```

or:

```
test:     MACRO
           IFC \1 "c"
            nop
            nop
           ENDIF
          ENDM
```

Tip: Use a comma to separate instruction operands.

Example:

```
MemBlock: DCB.B 8,$00
```

or:

```
test:     MACRO
           IFC \1,"c"
            nop
            nop
           ENDIF
          ENDM
```

12.3.46 A13007: Relative Branch with Illegal Target

Type: Error

Description: The offset specified in a PC relative addressing mode is a complex relocatable expression, a symbol defined in another section, or an externally defined symbol.

Example:

```
           XDEF Entry
           XREF MySubRoutine
DataSec:   SECTION
Data:      DS.B  1
Code1Sec:  SECTION
Entry1:
           NOP
           LDA #$60
           STA Data
CodeSec:   SECTION
Entry:
           LDA Data
           CMP #$60
           NOP
           BNE Entry1
           NOP
           BSR MySubRoutine
           NOP
main:      BRA main
```

Tip: The assembler does not support complex relocatable expressions. If a branch on an externally defined symbol or symbol defined in another section is needed, use the JMP or JSR instruction. If the branch label and instruction are located in the same assembly module, define the instruction and corresponding label in the same assembly SECTION.

Example:

```
                  XDEF Entry
                XREF MySubRoutine
      DataSec:  SECTION
      Data:     DS.B  1
      CodeSec: SECTION
      Entry1:
                NOP
                LDA #$60
                STA Data
      Entry:
                LDA Data
                CMP #$60
                NOP
                BNE Entry1
                NOP
                JSR MySubRoutine
                NOP
main:       BRA main
```

12.3.47 A13008: Illegal Expression

Type: Error

Description: An illegal expression is specified in a PC relative addressing mode.

Example:

```
CodeSec: SECTION
Entry:
         BRA #$200
```

Tip: Change the expression to a valid expression.

12.3.48 A13101: Illegal Operand Format

Type: Error

Description: An operand in the instruction is using an invalid addressing mode.

Example: This code generates an error message.

```
Entry:
         ADC X+
```

Tip: Use an allowed addressing mode for the instruction.

Example:

```
Entry:
         ADC ,X
         ADC X
         ADC #$5
```

12.3.49 A13102: Operand not Allowed

Type: Error

Description: This error message is issued for instructions BCLR or BRSET when the operand is not DIRECT or EXTENDED.

Example:

```
Entry:
         BRCLR 7, X
         BRCLR 7, SP
         BSET  7, X
```

Tip: Use an allowed addressing mode for the instruction.

12.3.50 A13106: Illegal Size Specification for HC05 Instruction

Type: Error

Description: A size operator follows an HC05 instruction. Size operators are coded as a period followed by a single character.

Example:

```
MyData: SECTION
data:   DS.B 1
MyCode: SECTION
entry:
        ADC.B data
        ADC.L data
        ADC.W data
        ADC.b data
        ADC.l data
        ADC.w data
```

Tip: Remove the size specification following the HC05 instruction.

Example:

```
MyData: SECTION
data:   DS.B 1
MyCode: SECTION
entry:
        ADC data
```

12.3.51 A13108: Illegal Character at the End of Line

Type: Error

Description: An invalid character or sequence of characters is detected at the end of an instruction. This message is generated if:

- A comment does not start with the comment character (;).
- An additional operand is specified in the instruction.

Example:

```
MyData: SECTION
data:   DS.B 1

MyCode: SECTION
entry:
        LDA data, #$7
        CLRA A
        CLR 0,X This is a comment
```

Tips:

- Remove invalid characters.
- Insert comment character at beginning of comment.
- Remove the extra operand.

Example:

```
MyData: SECTION
data:   DS.B 1

MyCode: SECTION
entry:
        LDA data
        CLRA
        CLR 0,X ;This is a comment
```

12.3.52 A13109: Positive Value Expected

Type: Error

Description: When using the instructions BSET, BCLR, BRSET or BRCLR, this error occurs if the specified value for the bit number is negative.

Example:

```
MyData: SECTION
data:   DS.B 1
NEG     EQU -2
MyCode: SECTION
entry:
        BCLR  -7, data
        BRCLR -4, data, entry
        BSET  -3, data
        BRSET NEG, data, entry
```

Tip: Use a positive value for the bit number.

Example:

```
MyData: SECTION
data:   DS.B 1
POS     EQU 2
MyCode: SECTION
entry:
        BCLR  7, data
        BRCLR 4, data, entry
        BSET  3, data
        BRSET POS, data, entry
```

12.3.53 A13110: Mask Expected

Type: Error

Description: When using the instructions BSET, BCLR, BRSET, or BRCLR, this error occurs if the specified value for the bit number is not DIRECT or EXTENDED.

Example:

```
MyData: SECTION
data:   DS.B 1

MyCode: SECTION
entry:
    BCLR  #$7, data
    BRCLR #$4, data, entry
    BRSET #$3, data, entry
    BSET  #$2, data
```

Tip: Use a correct value for the bit number 0, 1, 2, 3, 4, 5, 6, or 7.

Example:

```
MyData: SECTION
data:   DS.B 1

MyCode: SECTION
entry:
    BCLR  7, data
        BRCLR 4, data, entry
        BSET  3, data
    BRSET 2, data, entry
```

12.3.54 A13111: Value Out of Range

Type: Warning

Description: When using the instructions BSET, BCLR, BRSET, or BRCLR, this error occurs if the specified value for the bit number is greater than seven.

Example:

```
MyData: SECTION
data:   DS.B 1

MyCode: SECTION
entry:
    BCLR   20, data
    BRCLR  70, data, entry
    BSET    9, data
    BRSET 200, data, entry
```

Tip: Use a correct value for the bit number: 0, 1, 2, 3, 4, 5, 6 or 7.

Example:

```
MyData: SECTION
data:   DS.B 1

MyCode: SECTION
entry:
    BCLR  7, data
        BRCLR 4, data, entry
        BSET  3, data
        BRSET 2, data, entry
```

12.3.55 A13201: Lexical Error in First or Second Field

Type: Error

Description: An incorrect assembly line is detected. This message is generated if:

- An assembly instruction or directive starts on column 1
- An invalid identifier has been detected in the assembly line label or instruction part. Characters allowed as the first character in an identifier are:
  A...Z, a...z, _,
  Characters allowed after the first character in a label, instruction or directive name are:
  A...Z, a...z, 0...9, _,

Example:

```
CodeSec: SECTION
  …
LDA  #$20
  …
@label:
  …
4label:
```

Tips: Depending on the reason why the message was generated, take these actions:

- Insert at least one space in front of the directive or instruction.
- Change the label, directive, or instruction name to a valid identifier.

Example:

```
CodeSec: SECTION
  …
  LDA  #$20

  …
_label:
  …
_4label:
```

12.3.56 A13202: Not an HC05 Instruction or Directive

Type: Error

Description: The identifier detected in an assembly line instruction is neither an assembly directive, HC05 instruction, nor user-defined macro.

Example:

```
CodeSec: SECTION
  …
   LDAA #$5510
```

Tip: Change the identifier to an assembly directive, HC05 instruction, or name of a user-defined macro.

12.3.57 A13401: Value Out of Range — 128...127

Type: Error

Description: The offset between the current PC and the label specified as the PC relative address is not in the range of a signed byte (smaller than –128 or bigger than 127). An 8-bit signed PC relative offset is expected with the following instructions:

- Branch instructions:
  BCC, BCS, BEQ, BHCC, BHCS, BHI, BHS, BIH, BLO, BLS, BLT, BLS, BMC, BMI, BMS, BNE, BPL, BRA, BRN, BSR
- Third operand in BRCLR and BRSET instructions

Example for branch instructions:

```
DataSec: SECTION
var1:    DS.B 1
var2:    DS.B 2
CodeSec: SECTION
entry:
         LDA  var1
         BNE  label
dummyBl: DCB.B 200, $9D
label    STA var2
```

Tip: If BRA or BSR is used, replace it with JMP or JSR. Otherwise, the conditional branch instructions should first branch on a jump instruction linked to the desired label.

Example:

```
DataSec: SECTION
var1:    DS.B 1
var2:    DS.B 2
CodeSec: SECTION
entry:   LDA  var1
         BNE  label
         BRA  dummyBl
label:   JMP label2
dummyBl: DCB.B 200, $9D
label2:  STA var2
```

12.3.58 A13403: Complex Relocatable Expression not Supported

Type: Error

Description: A complex relocatable expression is detected when the expression contains:

- An operation between labels located in two different sections
- A multiplication, division, or modulo operation between two labels
- The addition of two labels located in the same section

Example:

```
DataSec1: SECTION
DataLbl1: DS.B 10
DataSec2: SECTION
DataLbl2: DS.B 15
offset: EQU DataLbl2 – DataLbl1
```

Tip: The assembler does not support complex relocatable expressions. The corresponding expression must be evaluated at execution time (works for labels located on page zero).

Example:

```
DataSec1: SECTION
DataLbl1: DS.B 10
DataSec2: SECTION
DataLbl2: DS.B 15
offset:   DS.B 1
....
MyCode:   SECTION
....
EvalOffset:
          LDA #DataLbl2
          SUB #DataLbl1
          STA offset
```

12.3.59 A13405: Code Size Per Section is Limited to 32 Kbytes

Type: Error

Description: A code or data section defined in the application is larger than 32 Kbytes.

Example:

```
cstSec:   SECTION
noptable: DCB.L 4000, $9D
          DCB.L 4000, $9D
          DCB.L 4000, $9D
          DCB.L 500, $9D
```

Tip: Split the section into sections smaller than 32 Kbytes. The order that the sections are allocated can be specified in the linker parameter file. Specify that the sections are allocated consecutively.

Example of an assembly file:

```
          XDEF entry
cstSec:   SECTION
noptbl:   DCB.L 4000, $9D
          DCB.L 4000, $9D
cstSec1:  SECTION
noptbl1:  DCB.L 4000, $9D
          DCB.L 500, $9D
entry:    BRA entry
```

Example of a parameter file:

```
LINK
  test.abs

NAMES test.o END
SECTIONS
    MY_RAM = READ_WRITE 0x0051 TO 0x00BF;
    MY_ROM = READ_ONLY  0x8301 TO 0x8DFD;
    ROM_2  = READ_ONLY  0xC000 TO 0xC1FD;
PLACEMENT
    DEFAULT_ROM     INTO MY_ROM;
    DEFAULT_RAM     INTO MY_RAM;
    cstSec, cstSec1 INTO ROM_2;
END
INIT entry
VECTOR ADDRESS 0xFFFE entry
```

12.3.60 A13406: HIGH with Initialized RAM not Supported

Type: Error

Description: The assembler does not support the use of the HIGH operator with initialized RAM.

Example:

```
MyData: SECTION
table:  DS.W 1
        DC.B low(table)
```

12.3.61 A13407: LOW with Initialized RAM not Supported

Type: Error

Description: The assembler does not support the use of the LOW operator with initialized RAM.

Example:

```
MyData: SECTION
table:  DS.W 1
        DC.B high(table)
```

12.3.62 A13601: Error in Expression

Type: Error

Description: An error has been detected by the assembler while reading the expression.

Example:

```
MyCode:  SECTION
entry:
                  BRA #$200
```

12.3.63 A13602: Error at End of Expression

Type:	Error
Description:	An error has been detected by the assembler at the end of the expression.
Example:	

```
label:  EQU 2*4)+6
```

Appendix A. MASM Compatibility

A.1 Contents

A.2 Introduction. 295
A.3 Comment Line 295
A.4 Constants. 296
A.5 Operators. 297
A.6 Directives 298

A.2 Introduction

The MCUez HC05 assembler has been extended to ensure compatibility with the MASM assembler.

A.3 Comment Line

A line starting with an asterisk (*) character is considered to be a comment line.

A.4 Constants

For compatibility with MASM, these integer constant notations are supported:

- Decimal constants are a sequence of decimal digits (0–9) followed by `d` or `D`.
- Hexadecimal constants are a sequence of hexadecimal digits (0–9, a–f, A–F) followed by `h` or `H`.
- Octal constants are a sequence of octal digits (0–7) followed by `o`, `O`, `q`, or `Q`.
- Binary constants are a sequence of binary digits (0–1) followed by `b` or `B`.

Example:

```
512d        ; decimal representation
512D        ; decimal representation
200h        ; hexadecimal representation
200H        ; hexadecimal representation
1000o       ; octal representation
1000O       ; octal representation
1000q       ; octal representation
1000Q       ; octal representation
1000000000b ; binary representation
1000000000B ; binary representation
```

A.5 Operators

For compatibility with the MASM assembler, the operator notation shown in **Table A-1** is supported.

Table A-1. Operators

Operator	Notation
Shift left	!<
Shift right	!>
Bitwise AND	!.
Bitwise OR	!+
Bitwise XOR	!x, !X

A.6 Directives

Table A-2 lists directives supported by MCUez for compatibility with MASM.

Table A-2. Directives

Operator	Notation	Description
RMB	DS	Defines storage for a variable
ELSEC	ELSE	Alternate of conditional block
ENDC	ENDIF	End of conditional block
NOL	NOLIST	Specifies that all subsequent instructions must not be inserted in the listing file
TTL	TITLE	Defines the user-defined title for the assembler listing file
GLOBAL	XDEF	Makes a symbol public (visible from outside)
PUBLIC	XDEF	Makes a symbol public (visible from outside)
EXTERNAL	XREF	Imports reference to an external symbol
XREFB	XREF.B	Imports reference to an external symbol located on the direct page
SWITCH		Allows switching to a section that has been previously defined
ASCT	ASCT: SECTION	Creates a predefined section with name id ASCT
BSCT	BSCT: SECTION SHORT	Creates a predefined section with name id BSCT. Variables defined in this section are accessed using the direct addressing mode.
CSCT	CSCT: SECTION	Creates a predefined section with name id CSCT
DSCT	DSCT: SECTION	Creates a predefined section with name id DSCT
IDSCT	IDSCT: SECTION	Creates a predefined section with name id IDSCT
IPSCT	IPSCT: SECTION	Creates a predefined section with name id IPSCT
PSCT	PSCT: SECTION	Creates a predefined section with name id PSCT

Appendix B. MCUasm Compatibility

B.1 Contents

B.2 Introduction 299
B.3 Labels 299
B.4 SET Directive 300
B.5 Obsolete Directives 300

B.2 Introduction

The MCUez HC05 assembler has been extended to ensure compatibility with the MCUasm assembler. MCUasm compatibility mode can be activated by specifying the option -MCUasm.

B.3 Labels

When MCUasm compatibility mode is activated, labels must be followed by a colon, even if they start on column one.

Example:

```
label:  NOP
```

B.4 SET Directive

When MCUasm compatibility mode is activated, relocatable expressions are allowed in a SET directive.

Example:

```
      label: SET *
```

If MCUasm compatibility mode is not activated, the SET label refers to absolute expressions.

B.5 Obsolete Directives

Table B-1 lists directives that are not recognized if MCUasm compatibility mode is activated.

Table B-1. Obsolete Directives

Operator	Notation	Description
RMB	DS	Defines storage for a variable
NOL	NOLIST	All subsequent instructions will not be inserted in the listing file.
TTL	TITLE	Defines title for assembler listing file
GLOBAL	XDEF	Makes a symbol public (visible from outside)
PUBLIC	XDEF	Makes a symbol public (visible from outside)
EXTERNAL	XREF	Imports reference to an external symbol

Appendix C. P&E Converter

C.1 Contents

C.2 Overview. 302

C.3 Input Files 303

C.4 Output Files. 303

C.5 Error Listing File. 303

C.6 Converter Environment. 303

C.7 Converting a P&E Assembler Source File 304
C.7.1 Window Title 305
C.7.2 Content Area 305
C.7.3 Toolbar. 306
C.7.4 Status Bar 307

C.8 Converter Menus. 307
C.8.1 File Menu 308
C.8.2 Converter Menu 309
C.8.3 View Menu. 310

C.9 Defining Environment Variables 310
C.9.1 Syntax for Variable Definition. 311
C.9.2 Specifying Paths in Variable Definition 311
C.9.3 Line Continuation in Variable Definition 311

C.10 Environment Variables 312
C.10.1 ENVIRONMENT 312
C.10.2 ERRORFILE 313
C.10.3 GENPATH. 314

C.11 Command Line Options 314
C.11.1 -H 315
C.11.2 -V 316

C.11.3 -O 316
C.11.4 -N 317

C.12 Ambiguities in P&E Assembler Syntax 317
C.12.1 Dashes in Labels 317
C.12.2 Percent Characters in Macros 319
C.12.3 Labels in Macros 320

C.13 Modifications During Conversion 321
C.13.1 Constants 321
C.13.2 Q Suffix 321
C.13.3 T Suffix 321

C.14 Default Base 321

C.15 Directives in the First Column 321

C.16 Expressions 322

C.17 Macro Declaration 322

C.18 Operators 323

C.19 Unsupported Commands 323

C.20 Supported Commands 324

C.2 Overview

This appendix describes the P&E to MCUez assembler converter. The converter is designed to convert P&E Microcomputer Systems Inc.'s assembler source files to MCUez assembler files.

The P&E to MCUez assembler converter offers a command line interface and an interactive interface. If an argument is not entered on the command line, a dialog box opens and prompts for the information.

C.3 Input Files

The converter requires a valid P&E assembler input file, but does not require a specific file extension. The default input file extension is *.S*.

The conversion process does not alter the input file. The P&E assembler syntax in the input file is converted to MCUez assembler syntax in a separate output file. Specific syntax conversions are described in **C.6 Converter Environment**.

C.4 Output Files

When a valid input file is converted successfully, the converter generates an output file in MCUez assembler syntax and saves the output file in the directory that contains the input file. The output file has the same name as the input file, but with the extension *.asm*.

C.5 Error Listing File

If the converter detects any errors, it will create an error listing file. The name and location of this file depends on settings in the environment variable ERRORFILE.

C.6 Converter Environment

The converter must be associated with a project directory to execute correctly. A project directory contains all files needed to configure a development environment.

During the MCUez installation process, the project directory is automatically set to *C:\MCUEZ\DEMO\HC05*. This default project directory contains initialization files that must be present in the MCUez environment.

C.7 Converting a P&E Assembler Source File

The following steps explain how to convert an example file to MCUez assembler syntax. The example file is named *test.s* and is located in the project directory.

1. Activate the **MCUez Shell**.
2. Select the **XLATE** icon from the shell toolbar (**Figure C-1**). When the converter is started, a **Tip of The Day** dialog is opened. This dialog contains the latest information. Click **Close** to close this dialog.

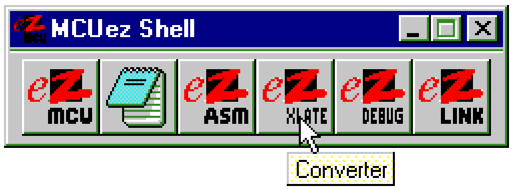
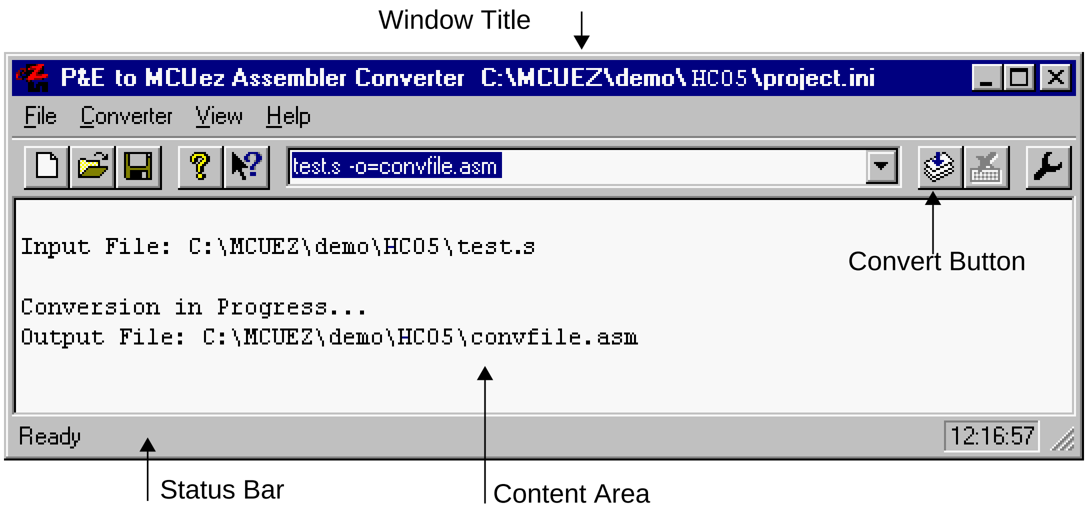


Figure C-1. XLATE Icon

3. Type in the P&E assembler source filename in the command line. Press **Enter** or the **Convert** button to start the conversion. By default, the output file contains the same name as the input file with a *.asm* extension. The user can specify the `-o` option on the command line to specify a new name for the output file. For example, `test.asm -o=convfile.asm`. **Figure C-2** shows an example of the conversion process.



```
Input File: C:\MCUEZ\demo\HC05\test.s

Conversion in Progress...
Output File: C:\MCUEZ\demo\HC05\convfile.asm
```

Figure C-2. Converter Window

4. The converter generates the *convfile.asm* file in the project directory. This file can be assembled using the MCUez assembler. When the output file location appears, the conversion has completed successfully.

C.7.1 Window Title

The window title displays the application name and project name. If no project is loaded, **Default Configuration** is displayed. An asterisk (*) after the configuration filename indicates that some values have changed.

C.7.2 Content Area

The content area displays information about the conversion. Information includes:

- Complete name, including path, of the file processed
- List of error, warning, and information messages
- Complete name, including path, of the converted file

NOTE: *A P&E source file also can be converted by dragging the source file from another application (for instance,* ***File Manager*** *or* ***Explorer****) and dropping it into the content area. The user also can drag and drop configuration files (.ini extension) to load a different configuration.*

Text in the converter window content area can have related information that consists of:

- Filename including a position inside the file
- Message number

Error feedback is performed automatically for some files listed in the content area. Double click on a message or filename in the content area to open the file in the editor specified during project configuration. The source file containing the error or warning message will open to the line containing the problem. The user can also select a line that contains a filename and click the right mouse button to display a menu that contains an **Open ..** entry (if file is editable).

A message number is displayed with message output. From this output, the corresponding help information can be opened three ways:

1. Select one line of the message and press F1. Help for the associated message number is displayed. If the selected line does not have a message number, the main help is displayed.
2. Press Shift-F1 and then click on the message text. If there is no associated message number, the main help is displayed.
3. Click the right mouse button on the message text and select **Help on ...**.

C.7.3 Toolbar

Figure C-3 illustrates the toolbar.

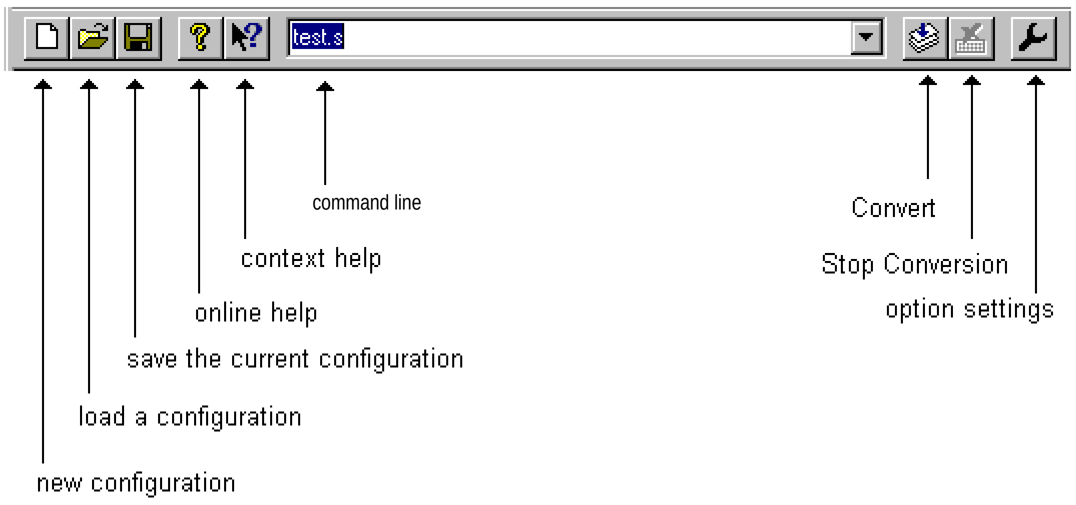


Figure C-3. Toolbar

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

The **Help** button opens the help file.

The **Context Help** button opens **Help** file information for a specific item. Click this button and select an item. Specific help is provided for menus, toolbar buttons, or window areas.

The editable combo box contains a list of commands executed. Once a command line has been selected or entered in this combo box, click the **Convert** button to execute this command. The **Stop Conversion** button aborts the current conversion.

The **Advanced Options** button opens the **Advanced Options** dialog box.

C.7.4 Status Bar

The status bar (**Figure C-4**) displays the current converter status or help information. Point to a menu entry or button in the toolbar to display a brief explanation in the message area.

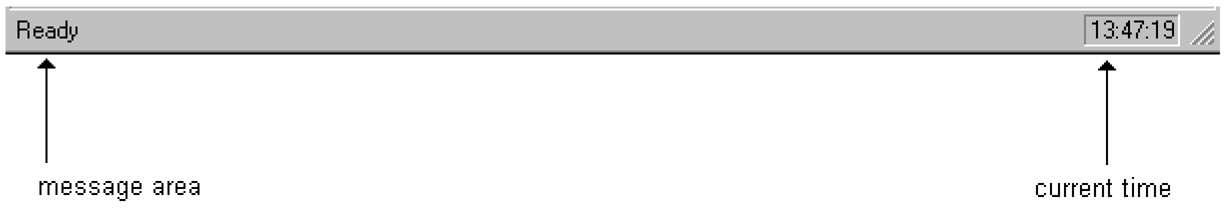


Figure C-4. Status Bar

C.8 Converter Menus

These four menus are available in the menu bar:

1. File — Contains entries to manage converter configuration files
2. Converter — Contains entries to set converter options
3. View — Contains entries to customize the converter window output
4. Help — Standard windows **Help** menu

C.8.1 File Menu

The **File** menu allows the user to convert a P&E assembler source file and save or load converter configuration files. A *.ini* configuration file contains this information:

- Converter option settings
- List of commands previously executed and the current command line
- Window position, size, and font
- Editor associated with the converter
- **Tip of the Day** settings

Configuration files are text files with a *.ini* extension. The user can define as many configuration files as required for a project. Load different configuration files by using the **File | Load Configuration** and **File | Save Configuration** menu entries or corresponding toolbar buttons.

Table C-1. File Menu Entries

Menu Entry	Description
Convert	Opens a standard **Open File** dialog box and displays a list of all files with extension *.s* located in the project directory. Select and open a file to begin the conversion process.
New/Default Configuration	Resets converter option settings to default values.
Load Configuration	Opens a standard **Open File** dialog box and displays a list of all files with extension *.ini* located in the project directory. Configuration data stored in the selected file is loaded and will be used by the next conversion.
Save Configuration	Saves the current settings
Save Configuration as...	Opens a standard **Save As** dialog box. Current settings are saved in the specified configuration file.
Configuration...	Opens the **Configuration** dialog box to specify the editor used for error feedback and type of information to be stored in the configuration file.
1. .... project.ini 2.	Recent project list. Select a file from this list to reopen a recently opened project file.
Exit	Closes the converter

C.8.2 Converter Menu

Use this menu to customize the converter.

Menu Entry	Description
Advanced	Defines options to be activated when converting an input file
Stop Converting	Stops the current conversion

Figure C-5 shows the **Advanced Options** dialog box. Use this dialog box to set and reset converter options.

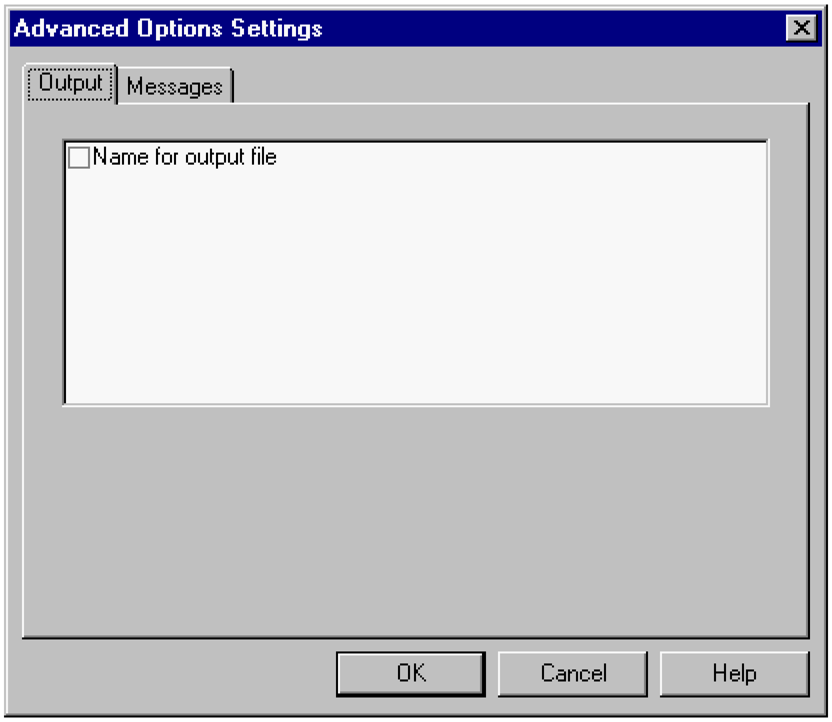


Figure C-5. Advanced Options Dialog Box

Group	Description
Output	Lists options related to generated output files
Messages	Lists options that control the generation of error messages

C.8.3 View Menu

This menu allows customization of the **Converter** window.

Menu entry	Description
Toolbar	Displays or hides toolbar in the converter window
Status Bar	Displays or hides status bar in the converter window
Log...	Customize output in the content area. Following entries are available when Log... is selected.
Change Font	Opens a standard font selection dialog box. Options selected in this dialog box are applied to the content area.
Clear Log	Clears the converter window content area.

C.9 Defining Environment Variables

This section explains the methods, syntax, paths, and line continuation symbols used to define an environment variable. Environment variable definitions are entered on the converter command line.

The three methods used to define environment variables are:

1. Use system environment variables supported by the operating system.
2. Put definitions in the project directory file (*project.ini*).
3. Put definitions in the file specified in the system environment variable ENVIRONMENT.

The converter first searches the system environment for environment variables, then the *project.ini* file, and finally the file specified in the ENVIRONMENT system variable. The converter uses a default definition for undefined environment variables.

The most effective method of defining environment variables is by putting them in a *project.ini* file in the project directory. The user can have a different *project.ini* file in separate directories. This allows the user to store a unique environment for separate projects.

C.9.1 Syntax for Variable Definition

The syntax used to define a variable is the environment name followed by an equal sign and the variable definition.

```
GENPATH=C:\INSTALL\LIB;D:\PROJECTS\TESTS;\usr\local\lib;
```

NOTE: *Do not include a space character before or after the equal sign.*

C.9.2 Specifying Paths in Variable Definition

Most environment variables include a path list of directory names separated by semicolons. The path list determines where the converter searches for project files.

The syntax used to create a path list is the environment name followed by the equal sign and directories.

```
GENPATH=C:\INSTALL\LIB;D:\PROJECT\TEST;
```

The converter searches the directories in the order they appear in the list.

If a directory name in the list is preceded by an asterisk (`*`), the converter searches that directory and its subdirectories.

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

C.9.3 Line Continuation in Variable Definition

The line continuation character (`\`) is used to define an environment variable over multiple lines in the *default.env* file.

Exercise caution when using this feature with paths.

```
GENPATH=.\
ERRORFILE=.\txt
```

Will be interpreted as:

```
GENPATH=.ERRORFILE=.\txt
```

To avoid such problems, use a semicolon (`;`) after the line continuation character.

```
GENPATH=.\;
ERRORFILE=.\txt
```

C.10 Environment Variables

This section describes how to define the environment variables that control the P&E to MCUez assembler converter. They are:

- ENVIRONMENT
- ERRORFILE
- GENPATH

C.10.1 ENVIRONMENT

Description:	Specifies the environment file for the MCUez assembler, linker, and debugger. The converter normally searches for the environment file (*default.env*) in the current directory. ENVIRONMENT allows the user to specify a different filename for the environment file.
Syntax:	`ENVIRONMENT=`<file>
Arguments:	<file>: Filename with path specification, without spaces
Example:	`ENVIRONMENT=\MCUEZ\prog\global.env`

NOTE: *ENVIRONMENT is a system-level environment variable and cannot be specified in a default environment file such as default.env.*

C.10.2 ERRORFILE

Tools:	Assembler and linker
Syntax:	`ERRORFILE=`<file>
Arguments:	<file>: Filename with possible format specifiers
Default:	EDOUT

Description: The environment variable ERRORFILE specifies the name of the error file.

Possible format specifiers are:

%n — Substitute with the filename, without path

%p — Substitute with the path of the source file

%f — Substitute with the full filename, including path (same as %p%n)

Example: *ERRORFILE=MyErrors.err* — Writes all errors to the file *MyErrors.err* in the project directory

ERRORFILE=\tmp\errors — Writes all errors to the file *errors* in the \tmp directory.

ERRORFILE=%f.err — Writes all errors to a file with the same name as the source file, but with extension *.err*. This file is placed in the same directory as the source file.

If the environment variable ERRORFILE is not set, errors are written to the default error file. The default error filename depends on the way the converter is started.

If no filename is provided on the converter command line, errors are written to the file *err.txt* in the project directory.

The following example uses this variable with the WinEdit editor. This editor looks for an error file called *EDOUT*.

Example:	Installation directory: *E:\INSTALL\PROG* Project sources: *D:\MEPHISTO* Common Sources for projects: *E:\CLIB* Entry in *default.env file (D:\MEPHISTO\DEFAULT.ENV):* *ERRORFILE=E:\INSTALL\PROG\EDOUT* Entry in *winedit.ini* file (in Windows directory): *OUTPUT=E:\INSTALL\PROG\EDOUT*

C.10.3 GENPATH

Description:	Defines paths to search for input files. The converter searches for input files in the project directory, then in the directories indicated by GENPATH. If a directory name in the path list is preceded by an asterisk (`*`), the converter searches that directory and its subdirectories.
Syntax:	`GENPATH=`<path>
Arguments:	<path>: Paths separated by semicolons, without spaces
Example:	`GENPATH=\obj;...\...\lib;`

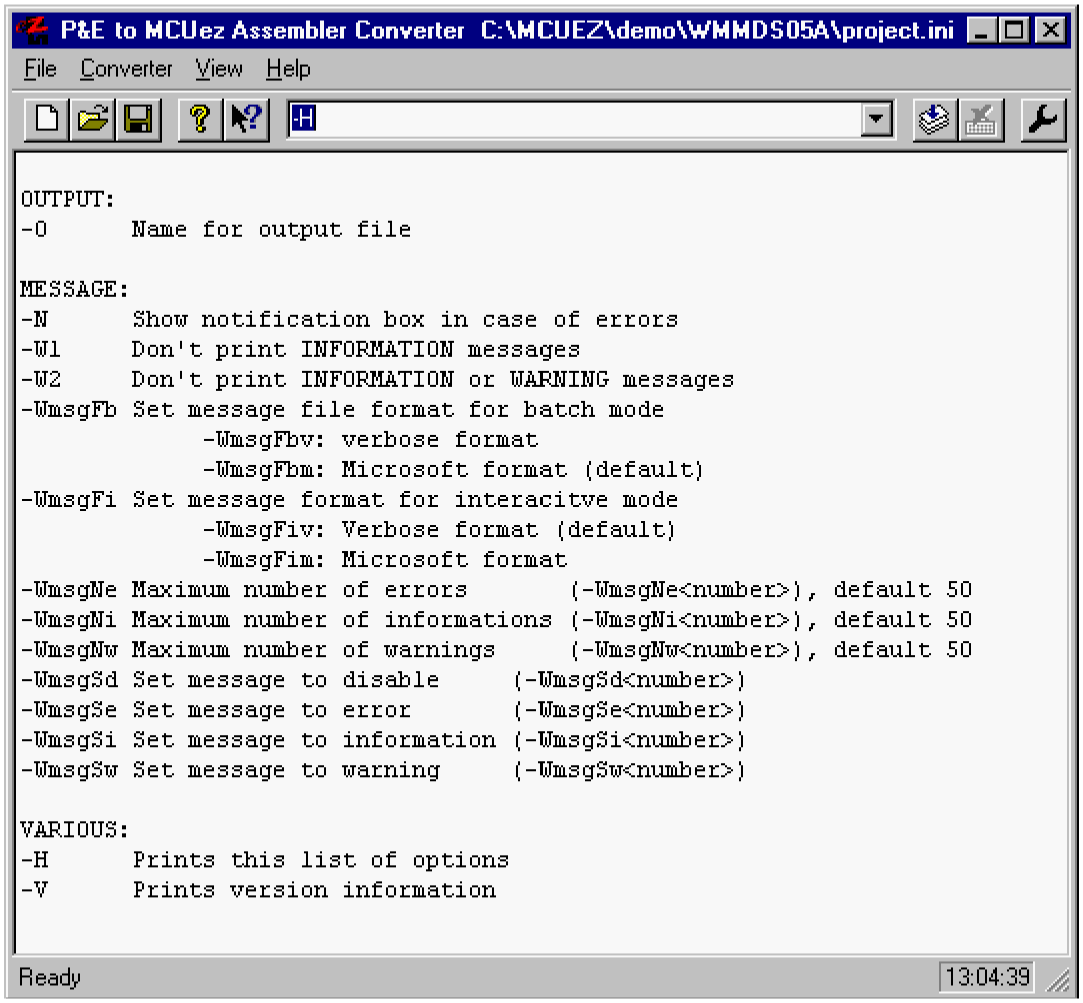
C.11 Command Line Options

The P&E to MCUez assembler converter offers a number of command line options to control the converter. These options are entered on the command line and consist of a dash (-) followed by an argument. The argument is made up of 1 to 128 letters and/or digits. Command line options are not case sensitive.

This section describes each command line option available for the converter.

C.11.1 -H

The `-H` (help) option displays the list of command line options with a short description.



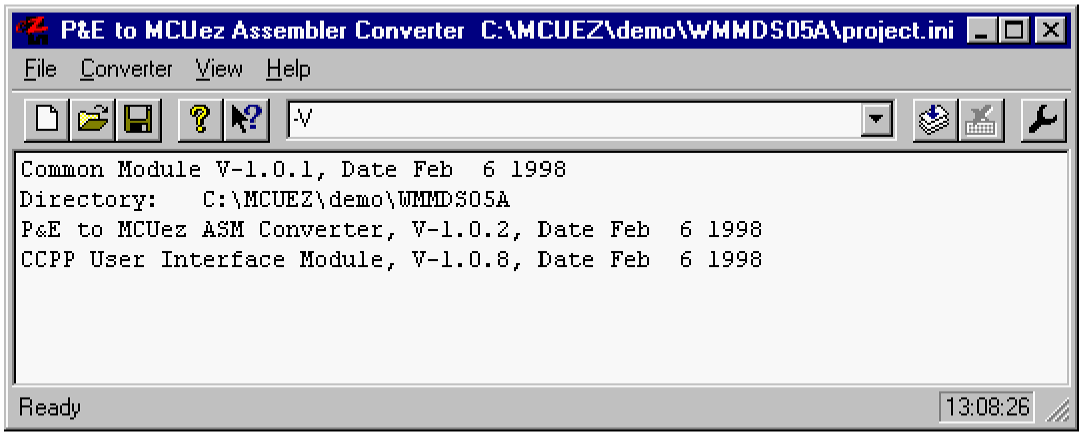
P&E to MCUez Assembler Converter C:\MCUEZ\demo\WMMDS05A\project.ini
File Converter View Help
-H

```
OUTPUT:
-O      Name for output file

MESSAGE:
-N      Show notification box in case of errors
-W1     Don't print INFORMATION messages
-W2     Don't print INFORMATION or WARNING messages
-WmsgFb Set message file format for batch mode
             -WmsgFbv: verbose format
             -WmsgFbm: Microsoft format (default)
-WmsgFi Set message format for interacitve mode
             -WmsgFiv: Verbose format (default)
             -WmsgFim: Microsoft format
-WmsgNe Maximum number of errors       (-WmsgNe<number>), default 50
-WmsgNi Maximum number of informations (-WmsgNi<number>), default 50
-WmsgNw Maximum number of warnings     (-WmsgNw<number>), default 50
-WmsgSd Set message to disable     (-WmsgSd<number>)
-WmsgSe Set message to error       (-WmsgSe<number>)
-WmsgSi Set message to information (-WmsgSi<number>)
-WmsgSw Set message to warning     (-WmsgSw<number>)

VARIOUS:
-H      Prints this list of options
-V      Prints version information
```

Figure C-6. Command Line Options

C.11.2 -V

The `-V` (version) option displays the converter version number and location of the project directory.

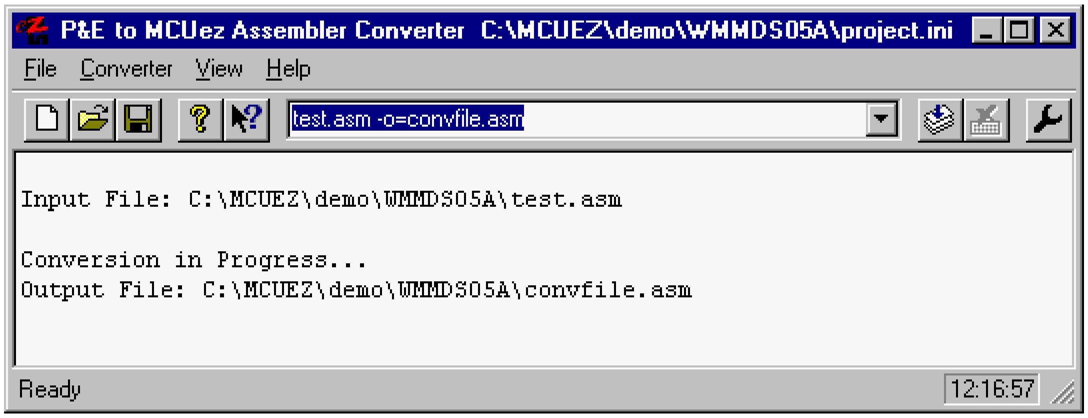

P&E to MCUez Assembler Converter C:\MCUEZ\demo\WMMDS05A\project.ini
File Converter View Help
-V
Ready
13:08:26

```
Common Module V-1.0.1, Date Feb  6 1998
Directory:   C:\MCUEZ\demo\WMMDS05A
P&E to MCUez ASM Converter, V-1.0.2, Date Feb  6 1998
CCPP User Interface Module, V-1.0.8, Date Feb  6 1998
```

Figure C-7. Version Number and Project Directory

C.11.3 -O

Syntax:	`-O=`<filename>
Arguments:	<filename>: Name of the MCUez assembler source file to be generated by the conversion
Description:	Defines the name of the output file


P&E to MCUez Assembler Converter C:\MCUEZ\demo\WMMDS05A\project.ini
File Converter View Help
test.asm -o=convfile.asm
Ready
12:16:57

```
Input File: C:\MCUEZ\demo\WMMDS05A\test.asm

Conversion in Progress...
Output File: C:\MCUEZ\demo\WMMDS05A\convfile.asm
```

Figure C-8. Output File Definition

C.11.4 -N

The `-N` (notify) option enables the converter to display an alert message if an error occurs during conversion. This is useful when running a makefile, since the converter waits for the user to acknowledge the message, thus suspending the makefile process.

C.12 Ambiguities in P&E Assembler Syntax

P&E assembler syntax contains three ambiguities that may affect the conversion process:

1. Dashes in labels
2. Percent characters in macros
3. Labels in macros

C.12.1 Dashes in Labels

Problem:

P&E assembler syntax allows the dash (`-`) character in label names. The presence of a dash can lead to ambiguity when converting label names. For example, `lab1-lab2` may refer to a label called `lab1-lab2` or two labels, `lab1` and `lab2`.

Solution:

If an expression contains a dash and begins on the first column of the input file, the converter assumes the expression is a label. In the output file, the converter substitutes the underscore character (`_`) for dashes used in labels and adds a comment. For example:

```
; CONV_01: Dash detected in Label
```

In the next example, the labels `lab-lab2` and `label2-3` need to be converted:

P&E input file:

```
$macro macro1
lab-lab2
  lda lab-lab2
  lda label2-3
  lda label2
$macroend

         ORG ROM
label1:
         lda label2-label3
         lda label2-3
label2:
         nop
label3:
         nop
label2-3:
         nop
```

MCUez output file:

```
macro1: MACRO
\@lab_lab2 ; CONV_01: Dash detected in Label. ;
CONV_03: Label detected in Macro.
  lda \@lab_lab2 ; CONV_01: Dash detected in Label.
; CONV_03:Label
  detected in Macro.
  lda label2_3 ; CONV_01: Dash detected in Label.
  lda label2
 ENDM

         ORG ROM
label1:
         lda label2-label3
          lda label2_3 ; CONV_01: Dash detected in
Label.
label2:
         nop
label3:
         nop
label2_3: ; CONV_01: Dash detected in Label.
         nop
```

The dash in the line `lda  label2-label3` does not change because the expression does not begin in the first column. The dash in the following line changes because the expression `label2-3` was defined in the previous lines.

C.12.2 Percent Characters in Macros

Problem: In macro definitions, the P&E assembler syntax perceives the percent character (%)as a macro parameter number. This creates ambiguity in the output file. In MCUez assembler syntax, an expression such as `label%2` may refer to appending the parameter number to the string `label` or the remainder of the variable (`label`) divided by two.

Solution: The converter replaces the percent character with the back slash (\) character and adds a comment. For example: `; CONV_02: Percent detected in Label`.

Therefore, the P&E assembler code:

```
#macro mymacro par1 par2 ; my comment
    lda #%1
    nop
    ldx %2
#macroend
```

Converts to the MCUez code:

```
mymacro: MACRO ;par1 par2 ; my comment
    lda #\1 ; CONV_02: Percent detected
         in Macro.
    nop
    ldx \2 ; CONV_02: Percent detected
         in Macro.
 ENDM
```

C.12.3 Labels in Macros

Problem: In P&E assembler syntax, a suffix is added to all labels defined within a macro. During conversion, the converter must determine whether to add a suffix to a label referenced in a macro definition.

Solution: The converter determines where the label is defined. If the label is defined inside a macro, the prefix `\@` is added to the label. This generates unique labels when the macro is expanded several times in a source file. The converter also adds a comment:
`; CONV_03: Label detected in Macro`.

In this P&E assembler code, the label `loop` is defined inside the macro:

```
#macro mymacro
     nop
loop:
      nop
      bra loop
#macroend
```

The converter converts this macro declaration to:

```
mymacro: MACRO
      nop
\@loop: ; CONV_03: Label detected in Macro.
      nop
      bra \@loop ; CONV_03: Label detected
          in Macro.
 ENDM
```

C.13 Modifications During Conversion

The following information explains syntax changes that are made during the conversion process.

C.13.1 Constants

The P&E assembler uses suffixes to indicate the type of number a constant represents. The MCUez assembler uses different suffixes to represent the same information.

C.13.2 Q Suffix

The suffix `Q` indicates a binary integer in P&E syntax. The converter substitutes the MCUez suffix `B` for the P&E suffix `Q`.

C.13.3 T Suffix

The suffix `T` indicates a decimal integer in P&E syntax. The converter substitutes the MCUez suffix `D` for the P&E suffix `T`.

C.14 Default Base

The P&E assembler uses base 16 as its default base number. The converter uses base 10 and inserts the directive `BASE $10` at the beginning of each output file.

C.15 Directives in the First Column

In P&E assembler syntax, directives start with a symbol: `/`, `#`, `$`, or `.` in the first column. In MCUez assembler syntax, only labels can begin in the first column.

To distinguish directives from labels, the converter replaces the symbol in front of each directive with a space character.

C.16 Expressions

Where the P&E assembler uses curly brackets { and }, the MCUez assembler uses parentheses (and). The converter simply replaces brackets with parentheses.

C.17 Macro Declaration

To define a macro with P&E assembler syntax, follow this format:

```
#macro mymacro
    nop
    nop
    nop
#macroend
```

The converter replaces the P&E macro definition directives with the appropriate MCUez assembler directives:

```
mymacro: MACRO
    nop
    nop
    nop
 ENDM
```

P&E assembler syntax also allows the user to specify parameters on the macro definition line:

```
#macro mymacro par1 par2 ; my comment
    nop
    nop
    nop
#macroend
```

The converter carries the parameters over to the converted code and places them in the appropriate position:

```
mymacro: MACRO ; par1 par2 ; my comment
    nop
    nop
    nop
 ENDM
```

C.18 Operators

The P&E assembler uses `<` as the shift left operator while the MCUez assembler uses `<<`. Similarly, the shift right operator in P&E syntax is `>` while it is `>>` in MCUez syntax.

The converter replaces `<` with `<<` and `>` with `>>`.

C.19 Unsupported Commands

The P&E assembler commands `cycle_adder_on`, `cycle_adder_off` and `subheader` are not supported by the MCUez assembler.

The converter replaces the symbols in front of these commands: `/, #, $,` or `.` with a semicolon `(;)`. The semicolon in the first column converts the commands into comments.

C.20 Supported Commands

The MCUez assembler supports many P&E assembler commands. The converter replaces these P&E commands with the matching MCUez commands.

Table C-2. Supported Commands

P&E Assembler Syntax	MCUez Assembler Syntax
DB	DC.B
DW	DC.W
EJECT	PAGE
ELSEIF	ELSE
HEADER	TITLE
IF	IFEQ
IFNOT	IFNE
PAGELENGTH	PLEN
PAGEWIDTH	PLEN
SET <label>	<label>: SET 1
SETNOT <label>	<label>: SET 0

Index

Symbols

- 81, 82
*. 144
.abs 73
.o 72, 74
.s1 73
.s2 73
.s3 73
.sx 73

A

ABSENTRY 152
Absolute Expression 144, 145
Absolute Section 109, 114
ABSPATH 73
Addressing Mode 125
Addressing Mode
  Direct. 125
  Extended 125
  Immediate 125
  Indexed, 16-bit offset 125
  Indexed, 8-bit offset 125
  Indexed, no offset 125
  Inherent 125
  Relative 125
ALIGN 152, 156, 168, 178
ASMOPTIONS 78
Assembler 50
  Input File. 71
  Output Files. 72

Assembler Menu 50

B

BASE 135, 152, 157

C

CLIST 153, 158
CODE 95
Code Generation 52
Code Section 108
Comment. 132
comment line. 118
Complex Relocatable Expression 144
Constant
  Binary 135, 296
  Decimal 135, 296
  Floating point 136
  Hexadecimal 135, 296
  Integer 135
  Octal 135, 296
  String. 136
Constant Section 108
Converter
  Configuration 308
Converter Menu 309
COPYRIGHT 67, 68, 69

D

Data Section 108
DC. 152, 160
DCB 152, 162
Debug File. 74, 175
DEFAULTDIR 72
Directive 125
  ABSENTRY 152
  ALIGN 152, 156, 168, 178

BASE . . . 135, 152, 157
CLIST . . . 153, 158
DC . . . 152, 160
DCB . . . 152, 162
DS . . . 152, 163
ELSE . . . 154, 164
ELSEC . . . 298
END . . . 152, 165
ENDC . . . 298
ENDIF . . . 154, 166
ENDM . . . 154, 166, 180
EQU . . . 133, 151, 167
EVEN . . . 152, 168
EXTERNAL . . . 298, 300
FAIL . . . 153, 169
GLOBAL . . . 298, 300
IF . . . 154, 172
IFC . . . 154, 174
IFDEF . . . 154, 174
IFEQ . . . 154, 174
IFGE . . . 154, 174
IFGT . . . 154, 174
IFLE . . . 154, 174
IFLT . . . 154, 174
IFNC . . . 154, 174
IFNDEF . . . 154, 174
IFNE . . . 154, 174
INCLUDE . . . 153, 175
LIST . . . 153, 175
LLEN . . . 153, 177
LONGEVEN . . . 153, 178
MACRO . . . 154, 179
MEXIT . . . 154, 180
MLIST . . . 153, 182
NOL . . . 298, 300
NOLIST . . . 153, 185
NOPAGE . . . 153, 186
ORG . . . 109, 151, 186

PAGE 153
PLEN. 153, 189
PUBLIC 298, 300
RMB 298, 300
SECTION 111, 151, 189
SET 133, 151, 191
SPC 153, 192
TABS 153, 192
TITLE 153, 192
TTL 298, 300
XDEF 134, 152, 193
XREF 134, 152, 194
XREFB 134, 152
Drag and Drop. 53
DS 152, 163

E

ELSE. 154, 164
ELSEC 298
END 152, 165
ENDC 298
ENDIF. 154, 166
ENDM. 154, 166, 180
-Env. 56
Environment
COPYRIGHT 67, 68, 69
File 56
INCLUDETIME 69
USERNAME. 67, 68
Variable. 56
Environment Variable 58
ABSPATH 61, 73
ASMOPTIONS 59
DEFAULTDIR 72
ENVIRONMENT 56
ERRORFILE. 65, 303
GENPATH 60, 72, 175

OBJPATH 62, 72
SRECORD 64, 73
TEXTPATH 63
EQU 133, 151, 167
Error feedback. 53
Error File. 74
Error Listing 74, 303
ERRORFILE. 303
EVEN 152, 168
Expression. 144
Absolute 144, 145
Complex Relocatable 144
Simple Relocatable 144, 146
EXTERNAL 298, 300
External Symbol 134

F

-FA2 83
FAIL 153, 169
File
Debug 74, 175
Environment 56
Error 74
Include. 72
Listing 73, 153, 175
Motorola S 73
Object 72
PRM 110, 112, 113
Source 72
File Menu 43, 308
Floating-Point Constant 136

G

GENPATH 72, 175
GLOBAL 298, 300
Graphical Interface 39

H

-H 84
-H Option 315
HIGH. 135
HOST 52, 82

I

IF 154, 172
IFC 154, 174
IFDEF 154, 174
IFEQ 154, 174
IFGE 154, 174
IFGT 154, 174
IFLE 154, 174
IFLT 154, 174
IFNC 154, 174
IFNDEF. 154, 174
IFNE 154, 174
INCLUDE 153, 175
Include Files 72
INCLUDETIME 69
INPUT 81
Input File 52
Instruction 120
Integer Constant 135

L

-L 106
Label 119
-Lc 87
-Ld 89
-Le 91
-Li 93
LIST 153, 175
Listing File 73, 153, 175
LLEN 153, 177
LONGEVEN. 153, 178

LOW 135

M

MACRO 154, 179
Macro 125
-Mb 95
MCUTOOLS.INI 44
Memory Model 52
Menu Bar 42, 307
MESSAGE 52, 97, 99, 100
Message
    ERROR 237
    FATAL 237
    WARNING 237
MEXIT 154, 180
MLIST 153, 182
Motorola S File 73

N

NOL 298, 300
NOLIST 153, 185
NOPAGE 153, 186

O

-O Option 316
Object File 72
OBJPATH 72
Operand 125
Operator 136, 297
    Addition 136, 143, 147
    Bitwise 138, 139, 143, 147, 297
    Division 137, 143, 147
    Force 142, 143
    HIGH 135
    Logical 139
    LOW 135, 141

Modulo 137, 143, 147
Multiplication 137, 143, 147
PAGE 135, 143
Precedence 142
Relational 140, 143
Shift. 138, 143, 147, 297
Sign 137, 143, 146
Subtraction 136, 143, 147
Option
CODE 95
-H 315
HOST 82
INPUT. 81
MESSAGE 97, 99, 100
-O 316
-V 316
VARIOUS. 84, 98
ORG 109, 151, 186
Output 52

P

PAGE 135, 153, 188
Path List 57
PLEN. 153, 189
PRM File. 110, 112, 113
PUBLIC 298, 300

R

Relocatable Section. 111, 114
Reserved Symbol 135
RMB 298, 300

S

SECTION 111, 151, 189
Section
  Absolute 109, 114
  Code 108
  Constant 108
  Data 108
  Relocatable 111, 114
SET 133, 151, 191
SHORT 190
Simple Relocatable Expression. 144, 146
Source File 72
source line 118
SPC 153, 192
Starting 38
Status Bar 42
String Constant 136
Symbol 132
  External 134
  Reserved 135
  Undefined 134
  User-Defined 133

T

TABS 153, 192
Tip of the Day 38
TITLE 153, 192
Toolbar 41
TTL 298, 300

U

Undefined Symbol 134
unit 104
User-Defined Symbol 133
USERNAME. 67, 68

V

-V 98
-V Option 316
Variable
    Environment 56
VARIOUS. 84, 98
View Menu 50, 310

W

-W2 100
Window. 39
WinEdit 66, 313
-WmsgFbm 104
-WmsgFim 106
-WmsgNe 99, 101
-WmsgNi. 102
-WmsgNw. 103

X

XDEF 134, 152, 193
XREF 134, 152, 194
XREFB 134, 152

Need to know more? That's ez, too.

Technical support for MCUez development tools is available through your regional Motorola office or by contacting:

Motorola, Inc.
6501 William Cannon Drive West
MD:OE17
Austin, Texas 78735
Phone (800) 521-6274
Fax (602) 437-1858
CRC@CRC.email.sps.mot.com

How to reach us:
USA/EUROPE/Locations Not Listed: Motorola Literature Distribution; P.O. Box 5405, Denver, Colorado 80217, 1–800–441–2447 or 1-303-675-2140.
Customer Focus Center: 1–800–521–6274
JAPAN: Motorola Japan Ltd.; SPD, Strategic Planning Office, 141, 4–32–1, Nishi–Gotanda, Shinagawa–ku, Tokyo, Japan, 03–5487–8488
ASIA/PACIFIC: Motorola Semiconductors H.K. Ltd., Silicon Harbour Centre, 2 Dal King Street, Tai Po Industrial Estate, Tai Po, New Territories,
Hong Kong, 852–26668334
Mfax™, Motorola Fax Back System: RMFAX0@email.sps.mot.com; http://sps.motorola.com/mfax/; TOUCHTONE, 1-602–244–8609;
US & Canada ONLY, 1–800–774–1848
HOME PAGE: http://motorola.com/sps/

MOTORO
Semiconductor Produ