

MSP430 Assembly Language Tools v 3.2

User's Guide



Literature Number: SLAU131D
June 2009

| | |
|---|-----------|
| Preface | 13 |
| 1 Introduction to the Software Development Tools | 17 |
| 1.1 Software Development Tools Overview | 18 |
| 1.2 Tools Descriptions..... | 19 |
| 2 Introduction to Object Modules | 21 |
| 2.1 Sections | 22 |
| 2.2 How the Assembler Handles Sections | 23 |
| 2.2.1 Uninitialized Sections | 23 |
| 2.2.2 Initialized Sections | 24 |
| 2.2.3 Named Sections | 25 |
| 2.2.4 Subsections | 25 |
| 2.2.5 Section Program Counters | 26 |
| 2.2.6 Using Sections Directives | 26 |
| 2.3 How the Linker Handles Sections..... | 29 |
| 2.3.1 Default Memory Allocation | 30 |
| 2.3.2 Placing Sections in the Memory Map..... | 30 |
| 2.4 Relocation | 31 |
| 2.5 Run-Time Relocation | 32 |
| 2.6 Loading a Program..... | 32 |
| 2.7 Symbols in an Object File | 33 |
| 2.7.1 External Symbols..... | 33 |
| 2.7.2 The Symbol Table..... | 33 |
| 3 Assembler Description | 35 |
| 3.1 Assembler Overview | 36 |
| 3.2 The Assembler's Role in the Software Development Flow | 37 |
| 3.3 Invoking the Assembler..... | 38 |
| 3.4 Naming Alternate Directories for Assembler Input | 39 |
| 3.4.1 Using the --include_path Assembler Option | 40 |
| 3.4.2 Using the MSP430_A_DIR Environment Variable | 40 |
| 3.5 Source Statement Format..... | 42 |
| 3.5.1 Label Field..... | 42 |
| 3.5.2 Mnemonic Field..... | 43 |
| 3.5.3 Operand Field..... | 43 |
| 3.5.4 Comment Field | 43 |
| 3.6 Constants..... | 43 |
| 3.6.1 Binary Integers..... | 44 |
| 3.6.2 Octal Integers | 44 |
| 3.6.3 Decimal Integers | 44 |
| 3.6.4 Hexadecimal Integers..... | 44 |
| 3.6.5 Character Constants | 45 |

| | | |
|----------|--|------------|
| 3.6.6 | Assembly-Time Constants | 45 |
| 3.7 | Character Strings..... | 45 |
| 3.8 | Symbols | 46 |
| 3.8.1 | Labels..... | 46 |
| 3.8.2 | Local Labels..... | 46 |
| 3.8.3 | Symbolic Constants..... | 48 |
| 3.8.4 | Defining Symbolic Constants (--asm_define Option) | 48 |
| 3.8.5 | Predefined Symbolic Constants | 49 |
| 3.8.6 | Substitution Symbols..... | 50 |
| 3.9 | Expressions | 50 |
| 3.9.1 | Operators..... | 51 |
| 3.9.2 | Expression Overflow and Underflow..... | 51 |
| 3.9.3 | Well-Defined Expressions..... | 51 |
| 3.9.4 | Conditional Expressions..... | 51 |
| 3.10 | Source Listings | 52 |
| 3.11 | Debugging Assembly Source | 54 |
| 3.12 | Cross-Reference Listings | 55 |
| 4 | Assembler Directives | 57 |
| 4.1 | Directives Summary..... | 58 |
| 4.2 | Directives That Define Sections | 61 |
| 4.3 | Directives That Initialize Constants | 63 |
| 4.4 | Directives That Perform Alignment and Reserve Space | 65 |
| 4.5 | Directives That Format the Output Listings | 66 |
| 4.6 | Directives That Reference Other Files | 67 |
| 4.7 | Directives That Enable Conditional Assembly..... | 67 |
| 4.8 | Directives That Define Structures | 68 |
| 4.9 | Directives That Define Symbols at Assembly Time..... | 68 |
| 4.10 | Miscellaneous Directives | 69 |
| 4.11 | Directives Reference..... | 70 |
| 5 | Macro Description | 119 |
| 5.1 | Using Macros..... | 120 |
| 5.2 | Defining Macros..... | 120 |
| 5.3 | Macro Parameters/Substitution Symbols | 122 |
| 5.3.1 | Directives That Define Substitution Symbols..... | 123 |
| 5.3.2 | Built-In Substitution Symbol Functions..... | 124 |
| 5.3.3 | Recursive Substitution Symbols | 125 |
| 5.3.4 | Forced Substitution | 125 |
| 5.3.5 | Accessing Individual Characters of Subscripted Substitution Symbols..... | 126 |
| 5.3.6 | Substitution Symbols as Local Variables in Macros | 127 |
| 5.4 | Macro Libraries..... | 128 |
| 5.5 | Using Conditional Assembly in Macros | 129 |
| 5.6 | Using Labels in Macros | 131 |
| 5.7 | Producing Messages in Macros | 132 |
| 5.8 | Using Directives to Format the Output Listing | 133 |
| 5.9 | Using Recursive and Nested Macros | 134 |
| 5.10 | Macro Directives Summary..... | 135 |
| 6 | Archiver Description | 137 |
| 6.1 | Archiver Overview | 138 |

| | | |
|----------|---|------------|
| 6.2 | The Archiver's Role in the Software Development Flow..... | 139 |
| 6.3 | Invoking the Archiver | 140 |
| 6.4 | Archiver Examples..... | 141 |
| 7 | Linker Description | 143 |
| 7.1 | Linker Overview | 144 |
| 7.2 | The Linker's Role in the Software Development Flow | 145 |
| 7.3 | Invoking the Linker..... | 146 |
| 7.4 | Linker Options | 147 |
| 7.4.1 | Wild Cards in File, Section, and Symbol Patterns..... | 148 |
| 7.4.2 | Relocation Capabilities (--absolute_exe and --relocatable Options) | 148 |
| 7.4.3 | Allocate Memory for Use by the Loader to Pass Arguments (--arg_size Option) | 150 |
| 7.4.4 | Compress DWARF Information (--compress_dwarf Option) | 150 |
| 7.4.5 | Control Linker Diagnostics..... | 150 |
| 7.4.6 | Disable Automatic Library Selection (--disable_auto_rts Option)..... | 151 |
| 7.4.7 | Disable Conditional Linking (--disable_clink Option) | 151 |
| 7.4.8 | Link Command File Preprocessing (--disable_pp, --define and --undefine Options) | 151 |
| 7.4.9 | Define an Entry Point (--entry_point Option) | 152 |
| 7.4.10 | Set Default Fill Value (--fill_value Option) | 152 |
| 7.4.11 | Generate List of Dead Functions (--generate_dead_funcs_list Option) | 152 |
| 7.4.12 | Using Function Subsections (--gen_func_subsections Option)..... | 153 |
| 7.4.13 | Define Heap Size (--heap_size Option)..... | 153 |
| 7.4.14 | Hiding Symbols | 153 |
| 7.4.15 | Alter the Library Search Algorithm (--library Option, --search_path Option, and MSP430_C_DIR Environment Variable)..... | 154 |
| 7.4.16 | Change Symbol Localization | 156 |
| 7.4.17 | Make a Symbol Global (--make_global Option) | 156 |
| 7.4.18 | Make All Global Symbols Static (--make_static Option) | 156 |
| 7.4.19 | Create a Map File (--map_file Option) | 157 |
| 7.4.20 | Managing Map File Contents (--mapfile_contents Option) | 158 |
| 7.4.21 | Disable Name Demangling (--no_demangle) | 159 |
| 7.4.22 | Disable Merge of Symbolic Debugging Information (--no_sym_merge Option) | 159 |
| 7.4.23 | Strip Symbolic Information (--no_sym_table Option)..... | 159 |
| 7.4.24 | Name an Output Module (--output_file Option) | 160 |
| 7.4.25 | C Language Options (--ram_model and --rom_model Options) | 160 |
| 7.4.26 | Exhaustively Read and Search Libraries (--reread_libs and --priority Options) | 160 |
| 7.4.27 | Create an Absolute Listing File (--run_abs Option) | 161 |
| 7.4.28 | Designate Header Path (--runtime Option) | 161 |
| 7.4.29 | Scan All Libraries for Duplicate Symbol Definitions (--scan_libraries)..... | 161 |
| 7.4.30 | Define Stack Size (--stack_size Option) | 161 |
| 7.4.31 | Enforce Strict Compatibility (--strict_compatibility Option) | 161 |
| 7.4.32 | Mapping of Symbols (--symbol_map Option) | 162 |
| 7.4.33 | Introduce an Unresolved Symbol (--undef_sym Option)..... | 162 |
| 7.4.34 | Replace Multiply Routine With Hardware Multiplier Routine (--use_hw_mpy)..... | 162 |
| 7.4.35 | Display a Message When an Undefined Output Section Is Created (--warn_sections Option) | 163 |
| 7.4.36 | Generate XML Link Information File (--xml_link_info Option)..... | 163 |
| 7.5 | Linker Command Files | 164 |
| 7.5.1 | Reserved Names in Linker Command Files..... | 165 |
| 7.5.2 | Constants in Linker Command Files | 165 |

| | | |
|--------|---|-----|
| 7.6 | Object Libraries | 166 |
| 7.7 | The MEMORY Directive | 167 |
| 7.7.1 | Default Memory Model | 167 |
| 7.7.2 | MEMORY Directive Syntax | 167 |
| 7.8 | The SECTIONS Directive | 169 |
| 7.8.1 | SECTIONS Directive Syntax | 169 |
| 7.8.2 | Allocation..... | 171 |
| 7.8.3 | Specifying Input Sections | 175 |
| 7.8.4 | Using Multi-Level Subsections | 177 |
| 7.8.5 | Allocation Using Multiple Memory Ranges | 178 |
| 7.8.6 | Automatic Splitting of Output Sections Among Non-Contiguous Memory Ranges | 178 |
| 7.8.7 | Allocating an Archive Member to an Output Section..... | 179 |
| 7.9 | Specifying a Section's Run-Time Address | 181 |
| 7.9.1 | Specifying Load and Run Addresses | 181 |
| 7.9.2 | Uninitialized Sections..... | 181 |
| 7.9.3 | Referring to the Load Address by Using the .label Directive..... | 182 |
| 7.10 | Using UNION and GROUP Statements | 183 |
| 7.10.1 | Overlaying Sections With the UNION Statement..... | 183 |
| 7.10.2 | Grouping Output Sections Together | 184 |
| 7.10.3 | Nesting UNIONS and GROUPs | 185 |
| 7.10.4 | Checking the Consistency of Allocators | 186 |
| 7.10.5 | Naming UNIONS and GROUPs..... | 186 |
| 7.11 | Special Section Types (DSECT, COPY, and NOLOAD) | 187 |
| 7.12 | Default Allocation Algorithm | 187 |
| 7.12.1 | How the Allocation Algorithm Creates Output Sections | 188 |
| 7.12.2 | Reducing Memory Fragmentation | 188 |
| 7.13 | Assigning Symbols at Link Time | 189 |
| 7.13.1 | Syntax of Assignment Statements..... | 189 |
| 7.13.2 | Assigning the SPC to a Symbol | 189 |
| 7.13.3 | Assignment Expressions..... | 190 |
| 7.13.4 | Symbols Defined by the Linker | 191 |
| 7.13.5 | Assigning Exact Start, End, and Size Values of a Section to a Symbol..... | 191 |
| 7.13.6 | Why the Dot Operator Does Not Always Work | 192 |
| 7.13.7 | Address and Dimension Operators..... | 192 |
| 7.14 | Creating and Filling Holes | 194 |
| 7.14.1 | Initialized and Uninitialized Sections | 194 |
| 7.14.2 | Creating Holes | 194 |
| 7.14.3 | Filling Holes | 196 |
| 7.14.4 | Explicit Initialization of Uninitialized Sections | 196 |
| 7.15 | Linker-Generated Copy Tables | 197 |
| 7.15.1 | A Current Boot-Loaded Application Development Process | 197 |
| 7.15.2 | An Alternative Approach | 197 |
| 7.15.3 | Overlay Management Example | 198 |
| 7.15.4 | Generating Copy Tables Automatically With the Linker | 198 |
| 7.15.5 | The table() Operator..... | 199 |
| 7.15.6 | Boot-Time Copy Tables..... | 200 |
| 7.15.7 | Using the table() Operator to Manage Object Components..... | 200 |
| 7.15.8 | Copy Table Contents..... | 201 |
| 7.15.9 | General Purpose Copy Routine..... | 202 |

| | | |
|-----------|---|------------|
| 7.15.10 | Linker-Generated Copy Table Sections and Symbols | 203 |
| 7.15.11 | Splitting Object Components and Overlay Management | 203 |
| 7.16 | Partial (Incremental) Linking..... | 205 |
| 7.17 | Linking C/C++ Code | 206 |
| 7.17.1 | Run-Time Initialization | 206 |
| 7.17.2 | Object Libraries and Run-Time Support | 206 |
| 7.17.3 | Setting the Size of the Stack and Heap Sections | 206 |
| 7.17.4 | Autoinitialization of Variables at Run Time | 207 |
| 7.17.5 | Initialization of Variables at Load Time | 207 |
| 7.17.6 | The --rom_model and --ram_model Linker Options..... | 208 |
| 7.18 | Linker Example..... | 209 |
| 8 | Absolute Lister Description | 213 |
| 8.1 | Producing an Absolute Listing | 214 |
| 8.2 | Invoking the Absolute Lister | 215 |
| 8.3 | Absolute Lister Example | 216 |
| 9 | Cross-Reference Lister Description | 219 |
| 9.1 | Producing a Cross-Reference Listing | 220 |
| 9.2 | Invoking the Cross-Reference Lister | 221 |
| 9.3 | Cross-Reference Listing Example | 222 |
| 10 | Object File Utilities Descriptions | 225 |
| 10.1 | Invoking the Object File Display Utility | 226 |
| 10.2 | Invoking the Disassembler..... | 227 |
| 10.3 | Invoking the Name Utility | 228 |
| 10.4 | Invoking the Strip Utility | 229 |
| 11 | Hex Conversion Utility Description | 231 |
| 11.1 | The Hex Conversion Utility's Role in the Software Development Flow..... | 232 |
| 11.2 | Invoking the Hex Conversion Utility | 233 |
| 11.2.1 | Invoking the Hex Conversion Utility From the Command Line | 233 |
| 11.2.2 | Invoking the Hex Conversion Utility With a Command File | 234 |
| 11.3 | Understanding Memory Widths | 236 |
| 11.3.1 | Target Width..... | 236 |
| 11.3.2 | Specifying the Memory Width | 237 |
| 11.3.3 | Partitioning Data Into Output Files | 238 |
| 11.3.4 | Specifying Word Order for Output Words | 239 |
| 11.4 | The ROMS Directive | 240 |
| 11.4.1 | When to Use the ROMS Directive..... | 241 |
| 11.4.2 | An Example of the ROMS Directive..... | 242 |
| 11.5 | The SECTIONS Directive..... | 243 |
| 11.6 | Excluding a Specified Section..... | 244 |
| 11.7 | Assigning Output Filenames | 245 |
| 11.8 | Image Mode and the -fill Option | 246 |
| 11.8.1 | Generating a Memory Image..... | 246 |
| 11.8.2 | Specifying a Fill Value | 246 |
| 11.8.3 | Steps to Follow in Using Image Mode | 246 |
| 11.9 | Controlling the ROM Device Address | 247 |
| 11.10 | Description of the Object Formats..... | 248 |
| 11.10.1 | ASCII-Hex Object Format (-a Option) | 248 |

| | | |
|-----------|--|------------|
| 11.10.2 | Intel MCS-86 Object Format (-i Option) | 249 |
| 11.10.3 | Motorola Exorciser Object Format (-m Option) | 250 |
| 11.10.4 | Texas Instruments SDSMAC (TI-Tagged) Object Format (-t Option) | 251 |
| 11.10.5 | TI-TXT Hex Format (--ti_txt Option) | 252 |
| 11.10.6 | Extended Tektronix Object Format (-x Option) | 253 |
| 12 | Sharing C/C++ Header Files With Assembly Source | 255 |
| 12.1 | Overview of the .cdecls Directive | 256 |
| 12.2 | Notes on C/C++ Conversions | 256 |
| 12.2.1 | Comments | 256 |
| 12.2.2 | Conditional Compilation (#if/#else/#ifdef/etc.) | 257 |
| 12.2.3 | Pragmas | 257 |
| 12.2.4 | The #error and #warning Directives | 257 |
| 12.2.5 | Predefined symbol __ASM_HEADER__ | 257 |
| 12.2.6 | Usage Within C/C++ asm() Statements | 257 |
| 12.2.7 | The #include Directive | 257 |
| 12.2.8 | Conversion of #define Macros | 257 |
| 12.2.9 | The #undef Directive | 258 |
| 12.2.10 | Enumerations | 258 |
| 12.2.11 | C Strings | 258 |
| 12.2.12 | C/C++ Built-In Functions | 259 |
| 12.2.13 | Structures and Unions | 259 |
| 12.2.14 | Function/Variable Prototypes | 259 |
| 12.2.15 | C Constant Suffixes | 260 |
| 12.2.16 | Basic C/C++ Types | 260 |
| 12.3 | Notes on C++ Specific Conversions | 260 |
| 12.3.1 | Name Mangling | 260 |
| 12.3.2 | Derived Classes | 260 |
| 12.3.3 | Templates | 261 |
| 12.3.4 | Virtual Functions | 261 |
| 12.4 | New Assembler Support | 261 |
| 12.4.1 | Enumerations (.enum/.emember/.endenum) | 261 |
| 12.4.2 | The .define Directive | 261 |
| 12.4.3 | The .undefine/.unasg Directives | 261 |
| 12.4.4 | The \$defined() Directive | 262 |
| 12.4.5 | The \$sizeof Built-In Function | 262 |
| 12.4.6 | Structure/Union Alignment & \$alignof() | 262 |
| 12.4.7 | The .cstring Directive | 262 |
| A | Symbolic Debugging Directives | 263 |
| A.1 | DWARF Debugging Format | 264 |
| A.2 | COFF Debugging Format | 264 |
| A.3 | Debug Directive Syntax | 265 |
| B | XML Link Information File Description | 267 |
| B.1 | XML Information File Element Types | 268 |
| B.2 | Document Elements | 268 |
| B.2.1 | Header Elements | 268 |
| B.2.2 | Input File List | 269 |
| B.2.3 | Object Component List | 270 |
| B.2.4 | Logical Group List | 271 |

| | | |
|----------|-----------------------|------------|
| B.2.5 | Placement Map | 273 |
| B.2.6 | Symbol Table..... | 274 |
| C | Glossary | 275 |

List of Figures

| | | |
|-------|--|-----|
| 1-1 | MSP430 Software Development Flow | 18 |
| 2-1 | Partitioning Memory Into Logical Blocks | 22 |
| 2-2 | Using Sections Directives Example | 27 |
| 2-3 | Object Code Generated by the File in Figure 2-2 | 28 |
| 2-4 | Combining Input Sections to Form an Executable Object Module | 30 |
| 3-1 | The Assembler in the MSP430 Software Development Flow | 37 |
| 3-2 | Example Assembler Listing | 53 |
| 4-1 | The .field Directive | 63 |
| 4-2 | Initialization Directives | 64 |
| 4-3 | The .align Directive | 65 |
| 4-4 | The .space and .bes Directives | 65 |
| 4-5 | Single-Precision Floating-Point Format | 83 |
| 4-6 | The .field Directive | 89 |
| 4-7 | The .usect Directive | 117 |
| 6-1 | The Archiver in the MSP430 Software Development Flow | 139 |
| 7-1 | The Linker in the MSP430 Software Development Flow | 145 |
| 7-2 | Section Allocation Defined by Example 7-4 | 171 |
| 7-3 | Run-Time Execution of Example 7-9 | 183 |
| 7-4 | Memory Allocation Shown in Example 7-11 and Example 7-12 | 184 |
| 7-5 | Autoinitialization at Run Time | 207 |
| 7-6 | Initialization at Load Time | 208 |
| 8-1 | Absolute Lister Development Flow | 214 |
| 9-1 | The Cross-Reference Lister in the MSP430 Software Development Flow | 220 |
| 11-1 | The Hex Conversion Utility in the MSP430 Software Development Flow | 232 |
| 11-2 | Hex Conversion Utility Process Flow | 236 |
| 11-3 | Object File Data and Memory Widths | 237 |
| 11-4 | Data, Memory, and ROM Widths | 239 |
| 11-5 | The infile.out File Partitioned Into Four Output Files | 242 |
| 11-6 | ASCII-Hex Object Format | 248 |
| 11-7 | Intel Hexadecimal Object Format | 249 |
| 11-8 | Motorola-S Format | 250 |
| 11-9 | TI-Tagged Object Format | 251 |
| 11-10 | TI-TXT Object Format | 252 |
| 11-11 | Extended Tektronix Object Format | 253 |

List of Tables

| | | |
|------|--|-----|
| 3-1 | MSP430 Assembler Options..... | 38 |
| 3-2 | Operators Used in Expressions (Precedence) | 51 |
| 3-3 | Symbol Attributes..... | 55 |
| 4-1 | Directives That Define Sections | 58 |
| 4-2 | Directives That Initialize Constants (Data and Memory) | 58 |
| 4-3 | Directives That Perform Alignment and Reserve Space | 59 |
| 4-4 | Directives That Format the Output Listing | 59 |
| 4-5 | Directives That Reference Other Files | 59 |
| 4-6 | Directives That Enable Conditional Assembly..... | 60 |
| 4-7 | Directives That Define Structures | 60 |
| 4-8 | Directives That Define Symbols at Assembly Time..... | 60 |
| 4-9 | Directives That Perform Miscellaneous Functions | 60 |
| 5-1 | Substitution Symbol Functions and Return Values..... | 124 |
| 5-2 | Creating Macros | 135 |
| 5-3 | Manipulating Substitution Symbols | 135 |
| 5-4 | Conditional Assembly | 135 |
| 5-5 | Producing Assembly-Time Messages..... | 135 |
| 5-6 | Formatting the Listing | 135 |
| 7-1 | Linker Options Summary | 147 |
| 7-2 | Groups of Operators Used in Expressions (Precedence) | 190 |
| 9-1 | Symbol Attributes in Cross-Reference Listing | 223 |
| 11-1 | Basic Hex Conversion Utility Options | 233 |
| 11-2 | Options for Specifying Hex Conversion Formats | 248 |
| A-1 | Symbolic Debugging Directives | 265 |

Read This First

About This Manual

The *MSP430 Assembly Language Tools User's Guide* explains how to use these assembly language tools:

- Assembler
- Archiver
- Linker
- Absolute lister
- Cross-reference lister
- Disassembler
- Object file display utility
- Name utility
- Strip utility
- Hex conversion utility

How to Use This Manual

This book helps you learn how to use the Texas Instruments assembly language tools designed specifically for the MSP430™ 16-bit devices. This book consists of four parts:

- **Introductory information**, consisting of [Chapter 1](#) and [Chapter 2](#), gives you an overview of the assembly language development tools. It also discusses object modules, which helps you to use the MSP430 tools more efficiently. Read [Chapter 2, Introduction to Object Modules](#), before using the assembler and linker.
- **Assembler description**, consisting of [Chapter 3](#) through [Chapter 5](#), contains detailed information about using the assembler. This portion explains how to invoke the assembler and discusses source statement format, valid constants and expressions, assembler output, and assembler directives. It also describes the macro language.
- **Additional assembly language tools description**, consisting of [Section 6.1](#) through [Chapter 11](#), describes in detail each of the tools provided with the assembler to help you create executable object files. For example, [Chapter 7](#) explains how to invoke the linker, how the linker operates, and how to use linker directives; [Chapter 11](#) explains how to use the hex conversion utility.
- **Reference material**, consisting of [Appendix A](#) through [Appendix C](#), provides supplementary information including symbolic debugging directives that the MSP430 C/C++ compiler uses. It also provides a description of the XML link information file and a glossary.

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a *special typeface*. Interactive displays use a bold version of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

Here is a sample of C code:

```
#include <stdio.h>
main() {printf("hello, cruel world\n");}
```

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** and parameters are in an *italic typeface*. Portions of a syntax that are in bold should be entered as shown; portions of a syntax that are in italics describe the type of information that should be entered.
- Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets. Unless the square brackets are in the **bold typeface**, do not enter the brackets themselves. The following is an example of a command that has an optional parameter:

```
cl430 [options] [filenames] [--run_linker [link_options] [object files]]
```

- Braces ({ and }) indicate that you must choose one of the parameters within the braces; you do not enter the braces themselves. This is an example of a command with braces that are not included in the actual syntax but indicate that you must specify either the --rom_model or --ram_model option:

```
cl430 --run_linker {--rom_model | --ram_model} filenames [--output_file=name.out]  
--library= libraryname
```

- In assembler syntax statements, column 1 is reserved for the first character of a label or symbol. If the label or symbol is optional, it is usually not shown. If it is a required parameter, it is shown starting against the left margin of the box, as in the example below. No instruction, command, directive, or parameter, other than a symbol or label, can begin in column 1.

```
symbol .usect "section name", size in bytes[, alignment]
```

- Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. This syntax is shown as [, ..., *parameter*].
- Following are other symbols and abbreviations used throughout this document:

| Symbol | Definition |
|---------------|------------------------------|
| B, b | Suffix — binary integer |
| H, h | Suffix — hexadecimal integer |
| LSB | Least significant bit |
| MSB | Most significant bit |
| 0x | Prefix — hexadecimal integer |
| Q, q | Suffix — octal integer |

Related Documentation From Texas Instruments

You can use the following books to supplement this user's guide:

[SLAU012](#) — **MSP430x3xx Family User's Guide**. Describes the MSP430x3xx™ CPU architecture, instruction set, pipeline, and interrupts for these ultra-low power microcontrollers.

[SLAU049](#) — **MSP430x1xx Family User's Guide**. Describes the MSP430x1xx™ CPU architecture, instruction set, pipeline, and interrupts for these ultra-low power microcontrollers.

[SLAU056](#) — **MSP430x4xx Family User's Guide**. Describes the MSP430x4xx™ CPU architecture, instruction set, pipeline, and interrupts for these ultra-low power microcontrollers.

[SLAU132](#) — **MSP430 Optimizing C/C++ Compiler User's Guide**. Describes the MSP430 C/C++ compiler. This C/C++ compiler accepts ANSI standard C/C++ source code and produces assembly language source code for the MSP430 devices.

[SLAU134](#) — **MSP430FE42x ESP30CE1 Peripheral Module User's Guide**. Describes common peripherals available on the MSP430FE42x and ESP430CE1 ultra-low power microcontrollers. This book includes information on the setup, operation, and registers of the ESP430CE1.

[SLAU144](#) — **MSP430x2xx Family User's Guide**. Describes the MSP430x2xx™ CPU architecture, instruction set, pipeline, and interrupts for these ultra-low power microcontrollers.

[SLAU208](#) — **MSP430x5xx Family User's Guide**. Describes the MSP430x5xx™ CPU architecture, instruction set, pipeline, and interrupts for these ultra-low power microcontrollers.

[SPRAAO8](#) — **Common Object File Format Application Report**. Provides supplementary information on the internal format of COFF object files. Much of this information pertains to the symbolic debugging information that is produced by the C compiler.

Introduction to the Software Development Tools

The MSP430™ is supported by a set of software development tools, which includes an optimizing C/C++ compiler, an assembler, a linker, and assorted utilities. This chapter provides an overview of these tools.

The MSP430 is supported by the following assembly language development tools:

- Assembler
- Archiver
- Linker
- Absolute lister
- Cross-reference lister
- Object file display utility
- Disassembler
- Name utility
- Strip utility
- Hex conversion utility

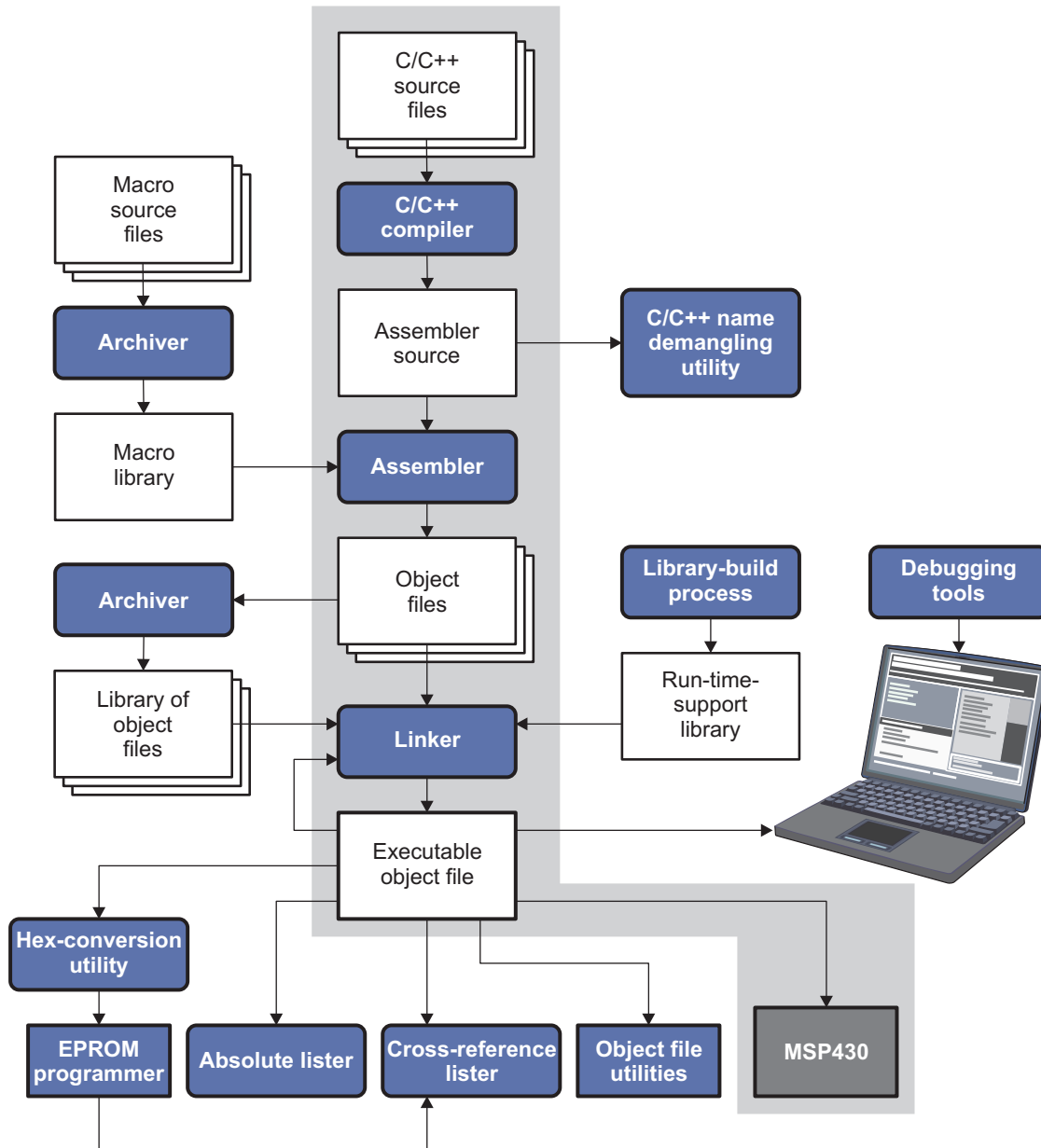
This chapter shows how these tools fit into the general software tools development flow and gives a brief description of each tool. For convenience, it also summarizes the C/C++ compiler and debugging tools. For detailed information on the compiler and debugger, and for complete descriptions of the MSP430, refer to books listed in *Related Documentation From Texas Instruments*.

| Topic | Page |
|---|-----------|
| 1.1 Software Development Tools Overview..... | 18 |
| 1.2 Tools Descriptions | 19 |

1.1 Software Development Tools Overview

Figure 1-1 shows the MSP430 software development flow. The shaded portion highlights the most common development path; the other portions are optional. The other portions are peripheral functions that enhance the development process.

Figure 1-1. MSP430 Software Development Flow



1.2 Tools Descriptions

The following list describes the tools that are shown in [Figure 1-1](#):

- The **C/C++ compiler** accepts C/C++ source code and produces MSP430 assembly language source code. A **shell program**, an **optimizer**, and an **interlist utility** are included in the compiler package:
 - The shell program enables you to compile, assemble, and link source modules in one step.
 - The optimizer modifies code to improve the efficiency of C/C++ programs.
 - The interlist utility interlists C/C++ source statements with assembly language output to correlate code produced by the compiler with your source code.

See the *MSP430 Optimizing C/C++ Compiler User's Guide* for more information.

- The **assembler** translates assembly language source files into machine language object modules. Source files can contain instructions, assembler directives, and macro directives. You can use assembler directives to control various aspects of the assembly process, such as the source listing format, data alignment, and section content. See [Chapter 3](#) through [Chapter 5](#). See the *MSP430x1xx Family User's Guide*, the *MSP430x2xx Family User's Guide*, the *MSP430x3xx Family User's Guide*, the *MSP430x4xx Family User's Guide*, or the *MSP430x5xx Family User's Guide* for detailed information on the assembly language instruction set.
- The **linker** combines object files into a single executable object module. As it creates the executable module, it performs relocation and resolves external references. The linker accepts relocatable object modules (created by the assembler) as input. It also accepts archiver library members and output modules created by a previous linker run. Link directives allow you to combine object file sections, bind sections or symbols to addresses or within memory ranges, and define or redefine global symbols. See [Chapter 7](#).
- The **archiver** allows you to collect a group of files into a single archive file, called a library. For example, you can collect several macros into a macro library. The assembler searches the library and uses the members that are called as macros by the source file. You can also use the archiver to collect a group of object files into an object library. The linker includes in the library the members that resolve external references during the link. The archiver allows you to modify a library by deleting, replacing, extracting, or adding members. See [Chapter 6](#).
- You can use the **library-build process** to build your own customized run-time-support library. See the *MSP430 Optimizing C/C++ Compiler User's Guide* for more information.
- The **hex conversion utility** converts an object file into TI-Tagged, ASCII-Hex, Intel, Motorola-S, or Tektronix object format. The converted file can be downloaded to an EPROM programmer. See [Chapter 11](#).
- The **absolute lister** uses linked object files to create .abs files. These files can be assembled to produce a listing of the absolute addresses of object code. See [Chapter 8](#).
- The **cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definition, and their references in the linked source files. See [Chapter 9](#).
- The main product of this development process is a module that can be executed in a **MSP430** device.

In addition, the following utilities are provided:

- The **object file display utility** prints the contents of object files, executable files, and/or archive libraries in both human readable and XML formats. See [Section 10.1](#).
- The **disassembler** writes the disassembled object code from object or executable files. See [Section 10.2](#).
- The **name utility** prints a list of names defined and referenced in a object or an executable file. See [Section 10.3](#).
- The **strip utility** removes symbol table and debugging information from object and executable files. See [Section 10.4](#).

Introduction to Object Modules

The assembler and linker create object modules that can be executed by a MSP430™ device.

Object modules make modular programming easier because they encourage you to think in terms of *blocks* of code and data when you write an assembly language program. These blocks are known as sections. Both the assembler and the linker provide directives that allow you to create and manipulate sections.

This chapter focuses on the concept and use of sections in assembly language programs.

| Topic | Page |
|--|------|
| 2.1 Sections..... | 22 |
| 2.2 How the Assembler Handles Sections | 23 |
| 2.3 How the Linker Handles Sections | 29 |
| 2.4 Relocation..... | 31 |
| 2.5 Run-Time Relocation | 32 |
| 2.6 Loading a Program..... | 32 |
| 2.7 Symbols in an Object File | 33 |

2.1 Sections

The smallest unit of an object file is called a *section*. A section is a block of code or data that occupies contiguous space in the memory map with other sections. Each section of an object file is separate and distinct. Object files usually contain three default sections:

| | |
|----------------------|--|
| .text section | usually contains executable code |
| .data section | usually contains initialized data |
| .bss section | usually reserves space for uninitialized variables |

In addition, the assembler and linker allow you to create, name, and link *named* sections that are used like the .data, .text, and .bss sections.

There are two basic types of sections:

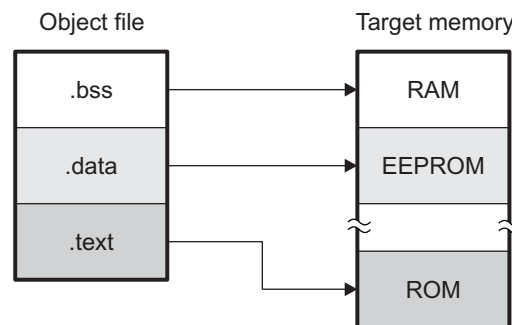
| | |
|-------------------------------|---|
| Initialized sections | contain data or code. The .text and .data sections are initialized; named sections created with the .sect assembler directive are also initialized. |
| Uninitialized sections | reserve space in the memory map for uninitialized data. The .bss section is uninitialized; named sections created with the .usect assembler directive are also uninitialized. |

Several assembler directives allow you to associate various portions of code and data with the appropriate sections. The assembler builds these sections during the assembly process, creating an object file organized as shown in [Figure 2-1](#).

One of the linker's functions is to relocate sections into the target system's memory map; this function is called *allocation*. Because most systems contain several types of memory, using sections can help you use target memory more efficiently. All sections are independently relocatable; you can place any section into any allocated block of target memory. For example, you can define a section that contains an initialization routine and then allocate the routine into a portion of the memory map that contains ROM.

[Figure 2-1](#) shows the relationship between sections in an object file and a hypothetical target memory.

Figure 2-1. Partitioning Memory Into Logical Blocks



2.2 How the Assembler Handles Sections

The assembler identifies the portions of an assembly language program that belong in a given section. The assembler has five directives that support this function:

- .bss
- .usect
- .text
- .data
- .sect

The .bss and .usect directives create *uninitialized sections*; the .text, .data, and .sect directives create *initialized sections*.

You can create subsections of any section to give you tighter control of the memory map. Subsections are created using the .sect and .usect directives. Subsections are identified with the base section name and a subsection name separated by a colon; see [Section 2.2.4](#).

Default Sections Directive

Note: If you do not use any of the sections directives, the assembler assembles everything into the .text section.

2.2.1 Uninitialized Sections

Uninitialized sections reserve space in MSP430 memory; they are usually allocated into RAM. These sections have no actual contents in the object file; they simply reserve memory. A program can use this space at run time for creating and storing variables.

Uninitialized data areas are built by using the .bss and .usect assembler directives.

- The .bss directive reserves space in the .bss section.
- The .usect directive reserves space in a specific uninitialized named section.

Each time you invoke the .bss or .usect directive, the assembler reserves additional space in the .bss or the named section. The syntaxes for these directives are:

| | |
|---------------|---|
| | .bss <i>symbol</i> , <i>size in bytes</i> [, <i>alignment</i>] |
| <i>symbol</i> | .usect " <i>section name</i> ", <i>size in bytes</i> [, <i>alignment</i>] |

| | |
|----------------------|---|
| <i>symbol</i> | points to the first byte reserved by this invocation of the .bss or .usect directive. The <i>symbol</i> corresponds to the name of the variable that you are reserving space for. It can be referenced by any other section and can also be declared as a global symbol (with the .global directive). |
| <i>size in bytes</i> | is an absolute expression. <ul style="list-style-type: none"> • The .bss directive reserves <i>size in bytes</i> bytes in the .bss section. The default value is 1 byte. • The .usect directive reserves <i>size in bytes</i> bytes in <i>section name</i>. You must specify a size; there is no default value. |
| <i>alignment</i> | is an optional parameter. It specifies the minimum alignment in bytes required by the space allocated. The default value is byte aligned. The value must be power of 2. |
| <i>section name</i> | tells the assembler which named section to reserve space in. See Section 2.2.3 . |

The initialized section directives (.text, .data, and .sect) tell the assembler to stop assembling into the current section and begin assembling into the indicated section. The .bss and .usect directives, however, *do not* end the current section and begin a new one; they simply escape from the current section temporarily. The .bss and .usect directives can appear anywhere in an initialized section without affecting its contents. For an example, see [Section 2.2.6](#).

The assembler treats uninitialized subsections (created with the .usect directive) in the same manner as uninitialized sections. See [Section 2.2.4](#), for more information on creating subsections.

2.2.2 *Initialized Sections*

Initialized sections contain executable code or initialized data. The contents of these sections are stored in the object file and placed in MSP430 memory when the program is loaded. Each initialized section is independently relocatable and may reference symbols that are defined in other sections. The linker automatically resolves these section-relative references.

Three directives tell the assembler to place code or data into a section. The syntaxes for these directives are:

| |
|---|
| <pre> .text .data .sect " <i>section name</i> " </pre> |
|---|

When the assembler encounters one of these directives, it stops assembling into the current section (acting as an implied end of current section command). It then assembles subsequent code into the designated section until it encounters another .text, .data, or .sect directive.

Sections are built through an iterative process. For example, when the assembler first encounters a .data directive, the .data section is empty. The statements following this first .data directive are assembled into the .data section (until the assembler encounters a .text or .sect directive). If the assembler encounters subsequent .data directives, it adds the statements following these .data directives to the statements already in the .data section. This creates a single .data section that can be allocated continuously into memory.

Initialized subsections are created with the .sect directive. The assembler treats initialized subsections in the same manner as initialized sections. See [Section 2.2.4](#), for more information on creating subsections.

2.2.3 Named Sections

Named sections are sections that *you* create. You can use them like the default `.text`, `.data`, and `.bss` sections, but they are assembled separately.

For example, repeated use of the `.text` directive builds up a single `.text` section in the object file. When linked, this `.text` section is allocated into memory as a single unit. Suppose there is a portion of executable code (perhaps an initialization routine) that you do not want allocated with `.text`. If you assemble this segment of code into a named section, it is assembled separately from `.text`, and you can allocate it into memory separately. You can also assemble initialized data that is separate from the `.data` section, and you can reserve space for uninitialized variables that is separate from the `.bss` section.

Two directives let you create named sections:

- The `.usect` directive creates uninitialized sections that are used like the `.bss` section. These sections reserve space in RAM for variables.
- The `.sect` directive creates initialized sections, like the default `.text` and `.data` sections, that can contain code or data. The `.sect` directive creates named sections with relocatable addresses.

The syntaxes for these directives are:

```
symbol    .usect "section name", size in bytes[, alignment ]
          .sect "section name"
```

The *section name* parameter is the name of the section. Section names are significant to 200 characters. You can create up to 32 767 separate named sections. For the `.usect` and `.sect` directives, a section name can refer to a subsection; see [Section 2.2.4](#) for details.

Each time you invoke one of these directives with a new name, you create a new named section. Each time you invoke one of these directives with a name that was already used, the assembler assembles code or data (or reserves space) into the section with that name. *You cannot use the same names with different directives.* That is, you cannot create a section with the `.usect` directive and then try to use the same section with `.sect`.

2.2.4 Subsections

Subsections are smaller sections within larger sections. Like sections, subsections can be manipulated by the linker. Subsections give you tighter control of the memory map. You can create subsections by using the `.sect` or `.usect` directive. The syntaxes for a subsection name are:

```
symbol    .usect "section name:subsection name", size in bytes[, alignment ]
          .sect "section name:subsection name"
```

A subsection is identified by the base section name followed by a colon and the name of the subsection. A subsection can be allocated separately or grouped with other sections using the same base name. For example, you create a subsection called `_func` within the `.text` section:

```
.sect ".text:_func"
```

Using the linker's `SECTIONS` directive, you can allocate `.text:_func` separately, or with all the `.text` sections. See [Section 7.8.1](#) for an example using subsections.

You can create two types of subsections:

- Initialized subsections are created using the `.sect` directive. See [Section 2.2.2](#).
- Uninitialized subsections are created using the `.usect` directive. See [Section 2.2.1](#).

Subsections are allocated in the same manner as sections. See [Section 7.8](#) for information on the `SECTIONS` directive.

2.2.5 Section Program Counters

The assembler maintains a separate program counter for each section. These program counters are known as *section program counters*, or *SPCs*.

An SPC represents the current address within a section of code or data. Initially, the assembler sets each SPC to 0. As the assembler fills a section with code or data, it increments the appropriate SPC. If you resume assembling into a section, the assembler remembers the appropriate SPC's previous value and continues incrementing the SPC from that value.

The assembler treats each section as if it began at address 0; the linker relocates each section according to its final location in the memory map. See [Section 2.4](#) for information on relocation.

2.2.6 Using Sections Directives

[Figure 2-2](#) shows how you can build sections incrementally, using the sections directives to swap back and forth between the different sections. You can use sections directives to begin assembling into a section for the first time, or to continue assembling into a section that already contains code. In the latter case, the assembler simply appends the new code to the code that is already in the section.

The format in [Figure 2-2](#) is a listing file. [Figure 2-2](#) shows how the SPCs are modified during assembly. A line in a listing file has four fields:

- Field 1** contains the source code line counter.
- Field 2** contains the section program counter.
- Field 3** contains the object code.
- Field 4** contains the original source statement.

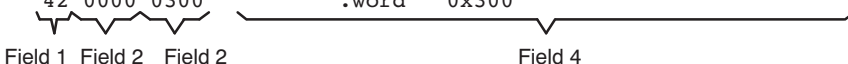
See [Section 3.10](#) for more information on interpreting the fields in a source listing.

Figure 2-2. Using Sections Directives Example

```

1          .global _mpyi
2          *****
3          * Assemble an initialized table into .data. *
4          *****
5 0000          .data
6 0000 0011  coeff  .word  011h,0x22,0x33
              0002 0022
              0004 0033
7          *****
8          * Reserve space in .bss for a variable. *
9          *****
10 0000          .bss  buffer,10
11          *****
12          * Still in .data. *
13          *****
14 0006 0123  ptr  .word  0x123
15          *****
16          * Assemble code into the .text section. *
17          *****
18 0000          .text
19 0000 403A  add:  MOV.W  #0x1234,R10
              0002 1234
20 0004 521A          ADD.W  &coeff+1,R10
              0006 0001!
21          *****
22          * Another initialized table into .data. *
23          *****
24 0008          .data
25 0008 00AA  ival  .word  0xAA,0xBB,0xCC
              000a 00BB
              000c 00CC
26          *****
27          * Define another section for more variables. *
28          *****
29 0000          var2  .usect "newvars", 1
30 0001          inbuf .usect "newvars", 7
31          *****
32          * Assemble more code into .text. *
33          *****
34 0008          .text
35 0008 403C  mpy:  MOV.W  #0x3456,R12
              000a 3456
36 000c 421D          MOV.W  &coeff,R13
              000e 0000!
37 0010 1290          CALL  _mpyi
              0012 FFEE!
38          *****
39          * Define a named section for int. vectors. *
40          *****
41 0000          .sect  "vectors"
42 0000 0300          .word  0x300

```



As Figure 2-3 shows, the file in Figure 2-2 creates five sections:

- .text** contains six 32-bit words of object code.
- .data** contains seven words of initialized data.
- vectors** is a named section created with the .sect directive; it contains one word of initialized data.
- .bss** reserves 10 bytes in memory.
- newvars** is a named section created with the .usect directive; it contains eight bytes in memory.

The second column shows the object code that is assembled into these sections; the first column shows the source statements that generated the object code.

Figure 2-3. Object Code Generated by the File in [Figure 2-2](#)

| Line numbers | Object code | Section |
|--------------|--------------------------------------|---------|
| 19 | 403A | .text |
| 19 | 1234 | |
| 20 | 521A | |
| 20 | 0001! | |
| 35 | 403C | |
| 35 | 3456 | |
| 36 | 421D | |
| 36 | 0000! | |
| 37 | 1290 | |
| 37 | FFEE! | |
| 6 | 0011 | .data |
| 6 | 0022 | |
| 6 | 0033 | |
| 14 | 0123 | |
| 25 | 00AA | |
| 25 | 00BB | |
| 25 | 00CC | |
| 10 | No data - ten bytes reserved | .bss |
| 29 | No data - eight bytes reserved | newvars |
| 30 | | |
| 42 | 0300 | vectors |

2.3 How the Linker Handles Sections

The linker has two main functions related to sections. First, the linker uses the sections in object files as building blocks; it combines input sections (when more than one file is being linked) to create output sections in an executable output module. Second, the linker chooses memory addresses for the output sections.

Two linker directives support these functions:

- The *MEMORY* directive allows you to define the memory map of a target system. You can name portions of memory and specify their starting addresses and their lengths.
- The *SECTIONS* directive tells the linker how to combine input sections into output sections and where to place these output sections in memory.

Subsections allow you to manipulate sections with greater precision. You can specify subsections with the linker's *SECTIONS* directive. If you do not specify a subsection explicitly, then the subsection is combined with the other sections with the same base section name.

It is not always necessary to use linker directives. If you do not use them, the linker uses the target processor's default allocation algorithm described in [Section 7.12](#). When you *do* use linker directives, you must specify them in a linker command file.

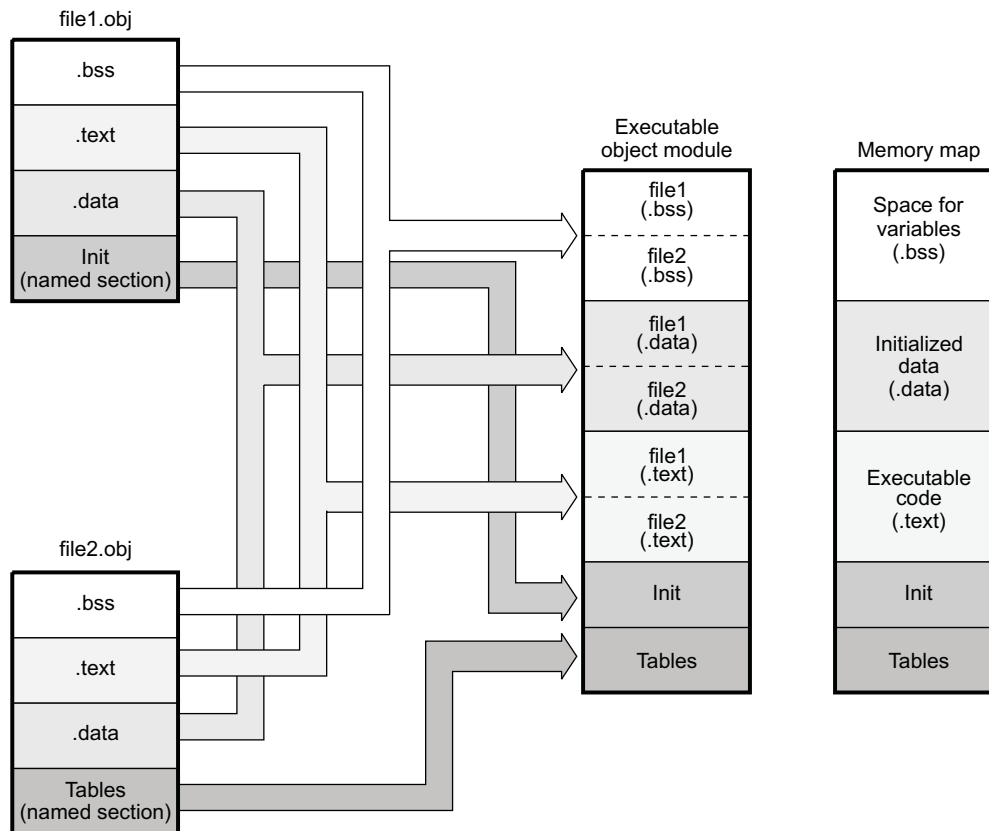
Refer to the following sections for more information about linker command files and linker directives:

- [Section 7.5](#), *Linker Command Files*
- [Section 7.7](#), *The MEMORY Directive*
- [Section 7.8](#), *The SECTIONS Directive*
- [Section 7.12](#), *Default Allocation Algorithm*

2.3.1 Default Memory Allocation

Figure 2-4 illustrates the process of linking two files together.

Figure 2-4. Combining Input Sections to Form an Executable Object Module



In Figure 2-4, file1.obj and file2.obj have been assembled to be used as linker input. Each contains the .text, .data, and .bss default sections; in addition, each contains a named section. The executable object module shows the combined sections. The linker combines the .text section from file1.obj and the .text section from file2.obj to form one .text section, then combines the two .data sections and the two .bss sections, and finally places the named sections at the end. The memory map shows how the sections are put into memory.

By default, the linker begins at 0h and places the sections one after the other in the following order: .text, .const, .data, .bss, .cinit, and then any named sections in the order they are encountered in the input files.

The C/C++ compiler uses the .const section to store string constants, and variables or arrays that are declared as *const*. The C/C++ compiler produces tables of data for autoinitializing global variables; these variables are stored in a named section called .cinit (see Example 7-7). For more information on the .const and .cinit sections, see the *MSP430 Optimizing C/C++ Compiler User's Guide*.

2.3.2 Placing Sections in the Memory Map

Figure 2-4 illustrates the linker's default method for combining sections. Sometimes you may not want to use the default setup. For example, you may not want all of the .text sections to be combined into a single .text section. Or you may want a named section placed where the .data section would normally be allocated. Most memory maps contain various types of memory (RAM, ROM, EPROM, etc.) in varying amounts; you may want to place a section in a specific type of memory.

For further explanation of section placement within the memory map, see the discussions in Section 7.7 and Section 7.8.

2.4 Relocation

The assembler treats each section as if it began at address 0. All relocatable symbols (labels) are relative to address 0 in their sections. Of course, all sections cannot actually begin at address 0 in memory, so the linker *relocates* sections by:

- Allocating them into the memory map so that they begin at the appropriate address as defined with the linker's MEMORY directive
- Adjusting symbol values to correspond to the new section addresses
- Adjusting references to relocated symbols to reflect the adjusted symbol values

The linker uses *relocation entries* to adjust references to symbol values. The assembler creates a relocation entry each time a relocatable symbol is referenced. The linker then uses these entries to patch the references after the symbols are relocated. [Example 2-1](#) contains a code segment for a MSP430 device that generates relocation entries.

Example 2-1. Code That Generates Relocation Entries

```

1          ** Generating Relocation Entries **
2
3          .ref X
4          .def Y
5
6 000000          .text
7 000000 5A0B          ADD.W   R10, R11
8 000002 4B82          MOV.W   R11, &X
9 000004 0000!
10 000006 4030          BR      #Y
11 000008 000A!
12 00000a 5B0C Y          ADD.W   R11, R12

```

In [Example 2-1](#), both symbols X and Y are relocatable. Y is defined in the .text section of this module; X is defined in another module. When the code is assembled, X has a value of 0 (the assembler assumes all undefined external symbols have values of 0), and Y has a value of 8 (relative to address 0 in the .text section). The assembler generates two relocation entries: one for X and one for Y. The reference to X is an external reference and the reference to Y is to an internally defined relocatable symbol (both are indicated by the ! character in the listing).

After the code is linked, suppose that X is relocated to address 0x0800. Suppose also that the .text section is relocated to begin at address 0x0600; Y now has a relocated value of 0x0608. The linker uses the relocation entry for the reference to X to patch the branch instruction in the object code:

```
4B820000!          becomes          4B820800
```

Each section in an object module has a table of relocation entries. The table contains one relocation entry for each relocatable reference in the section. The linker usually removes relocation entries after it uses them. This prevents the output file from being relocated again (if it is relinked or when it is loaded). A file that contains no relocation entries is an *absolute* file (all its addresses are absolute addresses). If you want the linker to retain relocation entries, invoke the linker with the --relocatable option (see [Section 7.4.2.2](#)).

2.5 Run-Time Relocation

At times you may want to load code into one area of memory and run it in another. For example, you may have performance-critical code in an external-memory-based system. The code must be loaded into external memory, but it would run faster in internal memory.

The linker provides a simple way to handle this. Using the `SECTIONS` directive, you can optionally direct the linker to allocate a section twice: first to set its load address and again to set its run address. Use the `load` keyword for the load address and the `run` keyword for the run address.

The load address determines where a loader places the raw data for the section. Any references to the section (such as references to labels in it) refer to its run address. The application must copy the section from its load address to its run address before the first reference of the symbol is encountered at run time; this does *not* happen automatically simply because you specify a separate run address. For an example that illustrates how to move a block of code at run time, see [Example 7-9](#).

If you provide only one allocation (either load or run) for a section, the section is allocated only once and loads and runs at the same address. If you provide both allocations, the section is actually allocated as if it were two separate sections of the same size.

Uninitialized sections (such as `.bss`) are not loaded, so the only significant address is the run address. The linker allocates uninitialized sections only once; if you specify both run and load addresses, the linker warns you and ignores the load address.

For a complete description of run-time relocation, see [Section 7.9](#).

2.6 Loading a Program

The linker produces executable object modules. An executable object module has the same format as object files that are used as linker input; the sections in an executable object module, however, are combined and relocated into target memory.

To run a program, the data in the executable object module must be transferred, or loaded, into target system memory. Several methods can be used for loading a program, depending on the execution environment. Common situations are described below:

- Code Composer Essentials can load an executable object module onto hardware. The Code Composer Essentials loader reads the executable file and copies the program into target memory.
- You can use the hex conversion utility (`hex430`, which is shipped as part of the assembly language package) to convert the executable object module into one of several object file formats. You can then use the converted file with an EPROM programmer to burn the program into an EPROM.

2.7 Symbols in an Object File

An object file contains a symbol table that stores information about symbols in the program. The linker uses this table when it performs relocation.

2.7.1 External Symbols

External symbols are symbols that are defined in one file and referenced in another file. You can use the `.def`, `.ref`, or `.global` directive to identify symbols as external:

| | |
|----------------|--|
| .def | The symbol is defined in the current file and used in another file. |
| .ref | The symbol is referenced in the current file, but defined in another file. |
| .global | The symbol can be either of the above. |

The following code segment illustrates these definitions.

```
x: ADD.W    #56, R11    ; Define x
    JMP     y          ; Reference y
    .global x          ; .def of x
    .global y          ; .ref of y
```

The `.global` directive for `x` declares that it is an external symbol defined in this module and that other modules can reference `x`. The `.global` directive for `y` declares that it is an undefined symbol that is defined in another module. The assembler determines that `y` is defined in another module because it is not defined in the current module.

The assembler places both `x` and `y` in the object file's symbol table. When the file is linked with other object files, the entry for `x` resolves references to `x` in other files. The entry for `y` causes the linker to look through the symbol tables of other files for `y`'s definition.

The linker must match all references with corresponding definitions. If the linker cannot find a symbol's definition, it prints an error message about the unresolved reference. This type of error prevents the linker from creating an executable object module.

2.7.2 The Symbol Table

The assembler always generates an entry in the symbol table when it encounters an external symbol (both definitions and references defined by one of the directives in [Section 2.7.1](#)). The assembler also creates special symbols that point to the beginning of each section; the linker uses these symbols to relocate references to other symbols.

The assembler does not usually create symbol table entries for any symbols other than those described above, because the linker does not use them. For example, labels are not included in the symbol table unless they are declared with the `.global` directive. For informational purposes, it is sometimes useful to have entries in the symbol table for each symbol in a program. To accomplish this, invoke the assembler with the `--output_all_syms` option (see [Section 3.3](#)).

Assembler Description

The MSP430™ assembler translates assembly language source files into machine language object files. These files are in object modules, which are discussed in [Chapter 2](#). Source files can contain the following assembly language elements:

| | |
|--------------------------------|---|
| Assembler directives | described in Chapter 4 |
| Macro directives | described in Chapter 5 |
| Assembly language instructions | described in the <i>MSP430x1xx Family User's Guide</i> , <i>MSP430x3xx Family User's Guide</i> , or <i>MSP430x4xx Family User's Guide</i> |

| Topic | Page |
|---|-----------|
| 3.1 Assembler Overview..... | 36 |
| 3.2 The Assembler's Role in the Software Development Flow..... | 37 |
| 3.3 Invoking the Assembler | 38 |
| 3.4 Naming Alternate Directories for Assembler Input | 39 |
| 3.5 Source Statement Format | 42 |
| 3.6 Constants | 43 |
| 3.7 Character Strings..... | 45 |
| 3.8 Symbols..... | 46 |
| 3.9 Expressions | 50 |
| 3.10 Source Listings..... | 52 |
| 3.11 Debugging Assembly Source | 54 |
| 3.12 Cross-Reference Listings..... | 55 |

3.1 Assembler Overview

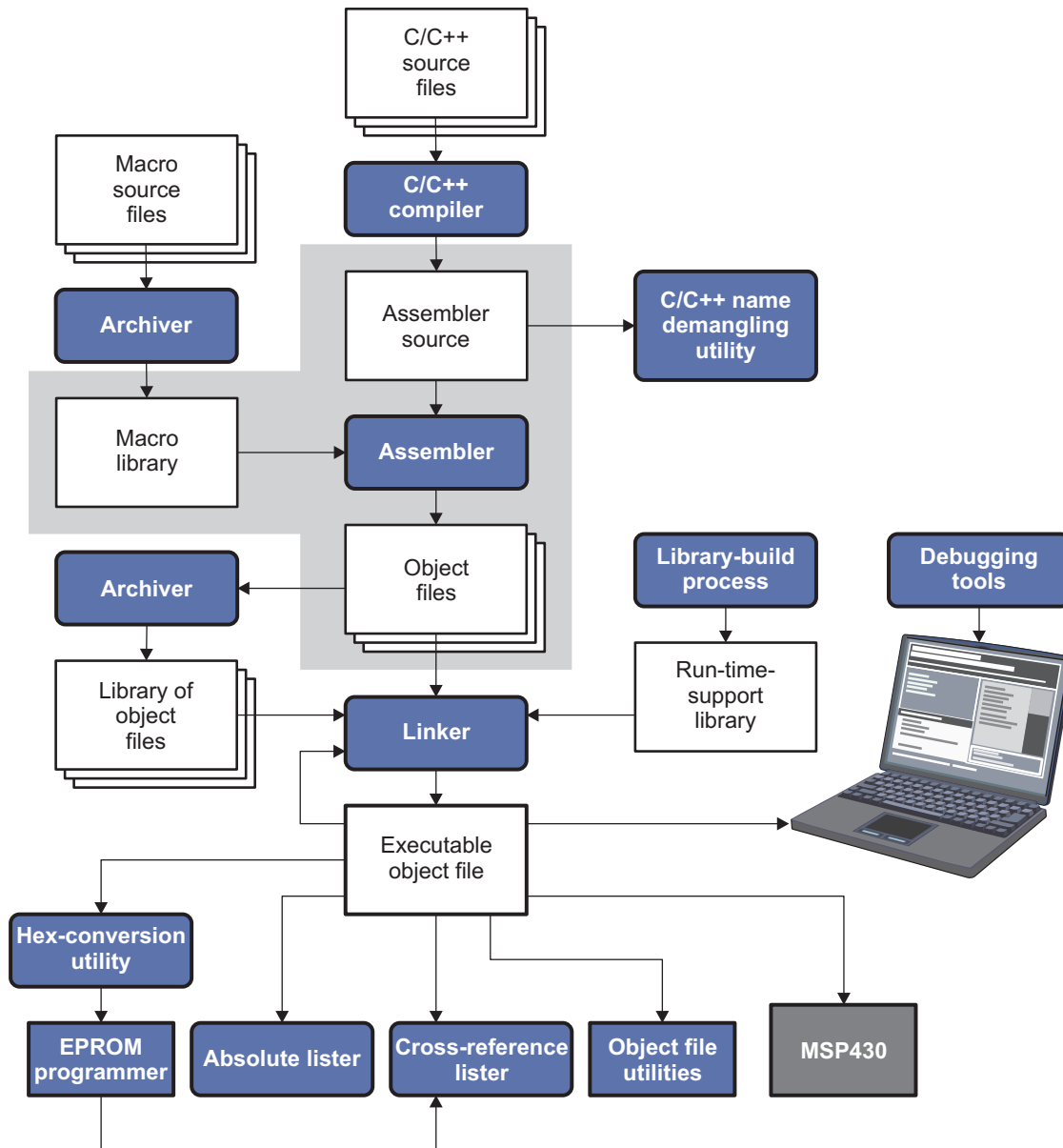
The 2-pass assembler does the following:

- Processes the source statements in a text file to produce a relocatable object file
- Produces a source listing (if requested) and provides you with control over this listing
- Allows you to segment your code into sections and maintain a section program counter (SPC) for each section of object code
- Defines and references global symbols and appends a cross-reference listing to the source listing (if requested)
- Allows conditional assembly
- Supports macros, allowing you to define macros inline or in a library

3.2 The Assembler's Role in the Software Development Flow

Figure 3-1 illustrates the assembler's role in the software development flow. The shaded portion highlights the most common assembler development path. The assembler accepts assembly language source files as input, both those you create and those created by the MSP430 C/C++ compiler.

Figure 3-1. The Assembler in the MSP430 Software Development Flow



3.3 Invoking the Assembler

To invoke the assembler, enter the following:

```
cl430 input file [options]
```

- cl430** is the command that invokes the assembler through the compiler. The compiler considers any file with an .asm extension to be an assembly file and calls the assembler.
- input file* names the assembly language source file.
- options* identify the assembler options that you want to use. Options are not case sensitive and can appear anywhere on the command line following the command. Precede each option with a hyphen.

The valid assembler options are listed in [Table 3-1](#):

Table 3-1. MSP430 Assembler Options

| Option | Alias | Description |
|--------------------------------|-------------|--|
| --absolute_listing | -aa | creates an absolute listing. When you use --absolute_listing, the assembler does not produce an object file. The --absolute_listing option is used in conjunction with the absolute_lister. |
| --asm_define=name[=def] | -ad | sets the <i>name</i> symbol. This is equivalent to defining <i>name</i> with a .set directive in the case of a numeric value or with an .asg directive otherwise. If <i>value</i> is omitted, the symbol is set to 1. See Section 3.8.4 . |
| --asm_dependency | -apd | performs preprocessing for assembly files, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility. The list is written to a file with the same name as the source file but with a .ppa extension. |
| --asm_includes | -api | performs preprocessing for assembly files, but instead of writing preprocessed output, writes a list of files included with the .include directive. The list is written to a file with the same name as the source file but with a .ppa extension. |
| --asm_listing | -al | produces a listing file with the same name as the input file with a .lst extension. |
| --asm_undefine=name | -au | undefines the predefined constant <i>name</i> , which overrides any --asm_define options for the specified constant. |
| --cmd_file=filename | -@ | appends the contents of a file to the command line. You can use this option to avoid limitations on command line length imposed by the host operating system. Use an asterisk or a semicolon (* or ;) at the beginning of a line in the command file to include comments. Comments that begin in any other column must begin with a semicolon. Within the command file, filenames or option parameters containing embedded spaces or hyphens must be surrounded with quotation marks. For example: "this-file.asm" |
| --copy_file=filename | -ahc | copies the specified file for the assembly module. The file is inserted before source file statements. The copied file appears in the assembly listing files. |
| --cross_reference | -ax | produces a cross-reference table and appends it to the end of the listing file; it also adds cross-reference information to the object file for use by the cross-reference utility. If you do not request a listing file but use the --cross_reference option, the assembler creates a listing file automatically, naming it with the same name as the input file with a .lst extension. |
| --include_file=filename | -ahi | includes the specified file for the assembly module. The file is included before source file statements. The included file does not appear in the assembly listing files. |
| --include_path=pathname | -I | specifies a directory where the assembler can find files named by the .copy, .include, or .mlib directives. There is no limit to the number of directories you can specify in this manner; each pathname must be preceded by the --include_path option. See Section 3.4.1 . |
| --output_all_syms | -as | puts all defined symbols in the object file's symbol table. The assembler usually puts only global symbols into the symbol table. When you use --output_all_syms, symbols defined as labels or as assembly-time constants are also placed in the table. |
| --quiet | -q | suppresses the banner and progress information (assembler runs in quiet mode). |

Table 3-1. MSP430 Assembler Options (continued)

| Option | Alias | Description |
|--|------------------|---|
| <code>--silicon_version={msp msp}</code> | | selects the instruction set version. Using <code>--silicon_version=msp</code> generates code for MSP430x devices (20-bit code addresses). Using <code>--silicon_version=m</code> generates code for 16-bit MSP430 devices. Modules assembled/compiled for 16-bit MSP devices are not compatible with modules that are assembled/compiled for 20-bit MSPx devices. The linker generates errors if an attempt is made to combine incompatible object files. |
| <code>--symdebug:dwarf</code> | <code>-g</code> | enables assembler source debugging in the C source debugger. Line information is output to the object module for every line of source in the assembly language source file. You cannot use the <code>--symdebug:dwarf</code> option on assembly code that contains <code>.line</code> directives. See Section 3.11 . |
| <code>--syms_ignore_case</code> | <code>-ac</code> | makes case insignificant in the assembly language files. For example, <code>--syms_ignore_case</code> makes the symbols ABC and abc equivalent. <i>If you do not use this option, case is significant</i> (default). Case significance is enforced primarily with symbol names, not with mnemonics and register names. |

3.4 Naming Alternate Directories for Assembler Input

The `.copy`, `.include`, and `.mlib` directives tell the assembler to use code from external files. The `.copy` and `.include` directives tell the assembler to read source statements from another file, and the `.mlib` directive names a library that contains macro functions. [Chapter 4](#) contains examples of the `.copy`, `.include`, and `.mlib` directives. The syntax for these directives is:

```
.copy ["]filename["]
.include ["]filename["]
.mlib ["]filename["]
```

The *filename* names a copy/include file that the assembler reads statements from or a macro library that contains macro definitions. If *filename* begins with a number the double quotes are required. The filename may be a complete pathname, a partial pathname, or a filename with no path information. The assembler searches for the file in the following locations in the order given:

1. The directory that contains the current source file. The current source file is the file being assembled when the `.copy`, `.include`, or `.mlib` directive is encountered.
2. Any directories named with the `--include_path` option
3. Any directories named with the `MSP430_A_DIR` environment variable
4. Any directories named with the `MSP430_C_DIR` environment variable

Because of this search hierarchy, you can augment the assembler's directory search algorithm by using the `--include_path` option (described in [Section 3.4.1](#)) or the `MSP430_A_DIR` environment variable (described in [Section 3.4.2](#)). `MSP430_C_DIR` is discussed in the *MSP430 Optimizing C/C++ Compiler User's Guide*.

3.4.1 Using the `--include_path` Assembler Option

The `--include_path` assembler option names an alternate directory that contains copy/include files or macro libraries. The format of the `--include_path` option is as follows:

```
cl430 --include_path=pathname source filename [other options]
```

There is no limit to the number of `--include_path` options per invocation; each `--include_path` option names one pathname. In assembly source, you can use the `.copy`, `.include`, or `.mlib` directive without specifying path information. If the assembler does not find the file in the directory that contains the current source file, it searches the paths designated by the `--include_path` options.

For example, assume that a file called `source.asm` is in the current directory; `source.asm` contains the following directive statement:

```
.copy "copy.asm"
```

Assume the following paths for the `copy.asm` file:

UNIX: /tools/files/copy.asm

Windows: c:\tools\files\copy.asm

You could set up the search path with the commands shown below:

| Operating System | Enter |
|---------------------|---|
| UNIX (Bourne shell) | <code>cl430 --include_path=/tools/files source.asm</code> |
| Windows | <code>cl430 --include_path=c:\tools\files source.asm</code> |

The assembler first searches for `copy.asm` in the current directory because `source.asm` is in the current directory. Then the assembler searches in the directory named with the `--include_path` option.

3.4.2 Using the `MSP430_A_DIR` Environment Variable

An environment variable is a system symbol that you define and assign a string to. The assembler uses the `MSP430_A_DIR` environment variable to name alternate directories that contain copy/include files or macro libraries.

The assembler looks for the `MSP430_A_DIR` environment variable and then reads and processes it. If the assembler does not find the `MSP430_A_DIR` variable, it then searches for `MSP430_C_DIR`. The processor-specific variables are useful when you are using Texas Instruments tools for different processors at the same time.

See the *MSP430 Optimizing C/C++ Compiler User's Guide* for details on `MSP430_C_DIR`.

The command syntax for assigning the environment variable is as follows:

| Operating System | Enter |
|---------------------|--|
| UNIX (Bourne Shell) | <code>MSP430_A_DIR="pathname₁;pathname₂; . . . "; export MSP430_A_DIR</code> |
| Windows | <code>set MSP430_A_DIR=pathname₁;pathname₂; . . .</code> |

The *pathnames* are directories that contain copy/include files or macro libraries. The pathnames must follow these constraints:

- Pathnames must be separated with a semicolon.
- Spaces or tabs at the beginning or end of a path are ignored. For example the space before and after the semicolon in the following is ignored:

```
set MSP430_A_DIR= c:\path\one\to\tools ; c:\path\two\to\tools
```


- Spaces and tabs are allowed within paths to accommodate Windows directories that contain spaces. For example, the pathnames in the following are valid:

```
set MSP430_A_DIR=c:\first path\to\tools;d:\second path\to\tools
```

In assembly source, you can use the `.copy`, `.include`, or `.mlib` directive without specifying path information. If the assembler does not find the file in the directory that contains the current source file or in directories named by the `--include_path` option, it searches the paths named by the environment variable.

For example, assume that a file called `source.asm` contains these statements:

```
.copy "copy1.asm"
.copy "copy2.asm"
```

Assume the following paths for the files:

UNIX: /`tools/files/copy1.asm` and `/dsys/copy2.asm`

Windows: `c:\tools\files\copy1.asm` and `c:\dsys\copy2.asm`

You could set up the search path with the commands shown below:

| Operating System | Enter |
|---------------------|---|
| UNIX (Bourne shell) | <code>MSP430_A_DIR="/dsys"; export MSP430_A_DIR cl430 --include_path=/tools/files source.asm</code> |
| Windows | <code>MSP430_A_DIR=c:\dsys cl430 --include_path=c:\tools\files source.asm</code> |

The assembler first searches for `copy1.asm` and `copy2.asm` in the current directory because `source.asm` is in the current directory. Then the assembler searches in the directory named with the `--include_path` option and finds `copy1.asm`. Finally, the assembler searches the directory named with `MSP430_A_DIR` and finds `copy2.asm`.

The environment variable remains set until you reboot the system or reset the variable by entering one of these commands:

| Operating System | Enter |
|---------------------|---------------------------------|
| UNIX (Bourne shell) | <code>unset MSP430_A_DIR</code> |
| Windows | <code>set MSP430_A_DIR=</code> |

3.5 Source Statement Format

MSP430 assembly language source programs consist of source statements that can contain assembler directives, assembly language instructions, macro directives, and comments. A source statement can contain four ordered fields (label, mnemonic, operand list, and comment). The general syntax for source statements is as follows:

```
[label[:]]mnemonic [operand list][:comment]
```

Following are examples of source statements:

```
SYM1      .set      2           ; Symbol SYM1 = 2.
Begin:    MOV.W     #SYM1, R11   ; Load R11 with 2.
          .word     016h        ; Initialize word (016h).
```

The MSP430 assembler reads up to 200 characters per line. Any characters beyond 200 are truncated. Keep the operational part of your source statements (that is, everything other than comments) less than 200 characters in length for correct assembly. Your comments can extend beyond the 200-character limit, but the truncated portion is not included in the listing file.

Follow these guidelines:

- All statements must begin with a label, a blank, an asterisk, or a semicolon.
- Labels are optional; if used, they must begin in column 1.
- One or more blanks must separate each field. Tab and space characters are blanks. You must separate the operand list from the preceding field with a blank.
- Comments are optional. Comments that begin in column 1 can begin with an asterisk or a semicolon (* or ;), but comments that begin in any other column *must* begin with a semicolon.
- A mnemonic cannot begin in column 1 or it will be interpreted as a label.

The following sections describe each of the fields.

3.5.1 Label Field

Labels are optional for all assembly language instructions and for most (but not all) assembler directives. When used, a label must begin in column 1 of a source statement. A label can contain up to 128 alphanumeric characters (A-Z, a-z, 0-9, _, and \$). Labels are case sensitive (except when the `--syms_ignore_case` option is used), and the first character cannot be a number. A label can be followed by a colon (:). The colon is not treated as part of the label name. If you do not use a label, the first character position must contain a blank, a semicolon, or an asterisk.

When you use a label, its value is the current value of the SPC. The label points to the statement it is associated with. For example, if you use the `.word` directive to initialize several words, a label points to the first word. In the following example, the label `Start` has the value 40h.

```
.      .      .      .
.      .      .      .
.      .      .      .
      9 0000      ; Assume some code was assembled
     10 0040 000A Start: .word 0Ah,3,7
           0044 0003
           0048 0007
```

A label on a line by itself is a valid statement. The label assigns the current value of the section program counter to the label; this is equivalent to the following directive statement:

```
label .equ $ ; $ provides the current value of the SPC
```

When a label appears on a line by itself, it points to the instruction on the next line (the SPC is not incremented):

```
      3 0050      Here:
      4 0050 0003      .word 3
```

If you do not use a label, the character in column 1 must be a blank, an asterisk, or a semicolon.

3.5.2 Mnemonic Field

The mnemonic field follows the label field. The mnemonic field cannot start in column 1; if it does, it is interpreted as a label. There is one exception: the parallel bars (||) of the mnemonic field can start in column 1. The mnemonic field can begin with one of the following items:

- Machine-instruction mnemonic (such as ADD, MOV, JMP)
- Assembler directive (such as .data, .list, .equ)
- Macro directive (such as .macro, .var, .mexit)
- Macro call

3.5.3 Operand Field

The operand field follows the mnemonic field and contains one or more operands. The operand field is not required for all instructions or directives. An operand consists of the following items:

- Symbols (see [Section 3.8](#))
- Constants (see [Section 3.6](#))
- Expressions (combination of constants and symbols; see [Section 3.9](#))

You must separate operands with commas.

You use immediate values as operands primarily with instructions. In some cases, you can use immediate values with the operands of directives. For instance, you can use immediate values with the .byte directive to load values into the current section. It is not usually necessary to use the # prefix for directives.

Compare the following statements:

```
ADD.W #10, R11

.byte 10
```

In the first statement, the # prefix is necessary to tell the assembler to add the value 10 to R1. In the second statement, however, the # prefix is not used; the assembler expects the operand to be a value and initializes a byte with the value 10.

See the *MSP430x1xx Family User's Guide*, the *MSP430x3xx Family User's Guide*, and the *MSP430x4xx Family User's Guide* for more information on the the syntax and usage of instructions. See [Chapter 4](#) for more information on the syntax and usage of directives.

3.5.4 Comment Field

A comment can begin in any column and extends to the end of the source line. A comment can contain any ASCII character, including blanks. Comments are printed in the assembly source listing, but they do not affect the assembly.

A source statement that contains only a comment is valid. If it begins in column 1, it can start with a semicolon (;) or an asterisk (*). Comments that begin anywhere else on the line must begin with a semicolon. The asterisk identifies a comment only if it appears in column 1.

3.6 Constants

The assembler supports several types of constants:

- Binary integer
- Octal integer
- Decimal integer
- Hexadecimal integer
- Character
- Assembly time

The assembler maintains each constant internally as a 32-bit quantity. Constants are not sign extended. For example, the constant 00FFh is equal to 00FF (base 16) or 255 (base 10); it *does not* equal -1. However, when used with the .byte directive, -1 is equivalent to 00FFh.

3.6.1 Binary Integers

A binary integer constant is a string of up to 32 binary digits (0s and 1s) followed by the suffix B (or b). If fewer than 32 digits are specified, the assembler right justifies the value and fills the unspecified bits with zeros. These are examples of valid binary constants:

| | |
|-----------|--|
| 00000000B | Constant equal to 0_{10} or 0_{16} |
| 0100000b | Constant equal to 32_{10} or 20_{16} |
| 01b | Constant equal to 1_{10} or 1_{16} |
| 11111000B | Constant equal to 248_{10} or $0F8_{16}$ |

3.6.2 Octal Integers

An octal integer constant is a string of up to 11 octal digits (0 through 7) followed by the suffix Q (or q). These are examples of valid octal constants:

| | |
|--------|---|
| 10Q | Constant equal to 8_{10} or 8_{16} |
| 010 | Constant equal to 8_{10} or 8_{16} format) |
| 10000Q | Constant equal to $32\ 768_{10}$ or 8000_{16} |
| 226q | Constant equal to 150_{10} or 96_{16} |

3.6.3 Decimal Integers

A decimal integer constant is a string of decimal digits ranging from -2147 483 648 to 4 294 967 295. These are examples of valid decimal constants:

| | |
|--------|--|
| 1000 | Constant equal to 1000_{10} or $3E8_{16}$ |
| -32768 | Constant equal to $-32\ 768_{10}$ or 8000_{16} |
| 25 | Constant equal to 25_{10} or 19_{16} |

3.6.4 Hexadecimal Integers

A hexadecimal integer constant is a string of up to eight hexadecimal digits followed by the suffix H (or h) or preceded by 0x. Hexadecimal digits include the decimal values 0-9 and the letters A-F or a-f. A *hexadecimal constant must begin with a decimal value (0-9)*. If fewer than eight hexadecimal digits are specified, the assembler right justifies the bits. These are examples of valid hexadecimal constants:

| | |
|-------|---|
| 78h | Constant equal to 120_{10} or 0078_{16} |
| 0x78 | Constant equal to 120_{10} or 0078_{16} format) |
| 0Fh | Constant equal to 15_{10} or $000F_{16}$ |
| 37ACh | Constant equal to $14\ 252_{10}$ or $37AC_{16}$ |

3.6.5 Character Constants

A character constant is a single character enclosed in *single* quotes. The characters are represented internally as 8-bit ASCII characters. Two consecutive single quotes are required to represent each single quote that is part of a character constant. A character constant consisting only of two single quotes is valid and is assigned the value 0. These are examples of valid character constants:

| | |
|-----|---|
| 'a' | Defines the character constant <i>a</i> and is represented internally as 61 ₁₆ |
| 'C' | Defines the character constant <i>C</i> and is represented internally as 43 ₁₆ |
| '' | Defines the character constant <i>'</i> and is represented internally as 27 ₁₆ |
| " | Defines a null character and is represented internally as 00 ₁₆ |

Notice the difference between character *constants* and character *strings* (Section 3.7 discusses character strings). A character constant represents a single integer value; a string is a sequence of characters.

3.6.6 Assembly-Time Constants

If you use the `.set` directive to assign a value to a symbol (see [Define Assembly-Time Constant](#)), the symbol becomes a constant. To use this constant in expressions, the value that is assigned to it must be absolute. For example:

```
shift3 .set 3
      MOV #shift3, R11
```

You can also use the `.set` directive to assign symbolic constants for register names. In this case, the symbol becomes a synonym for the register:

```
OP1 .set R11
    MOV OP1, 2(SP)
```

3.7 Character Strings

A character string is a string of characters enclosed in *double* quotes. Double quotes that are part of character strings are represented by two consecutive double quotes. The maximum length of a string varies and is defined for each directive that requires a character string. Characters are represented internally as 8-bit ASCII characters.

These are examples of valid character strings:

| | |
|-------------------------|---|
| "sample program" | defines the 14-character string <i>sample program</i> . |
| "PLAN ""C""" | defines the 8-character string <i>PLAN "C"</i> . |

Character strings are used for the following:

- Filenames, as in `.copy "filename"`
- Section names, as in `.sect "section name"`
- Data initialization directives, as in `.byte "charstring"`
- Operands of `.string` directives

3.8 Symbols

Symbols are used as labels, constants, and substitution symbols. A symbol name is a string of up to 200 alphanumeric characters (A-Z, a-z, 0-9, \$, and `_`). The first character in a symbol cannot be a number, and symbols cannot contain embedded blanks. The symbols you define are case sensitive; for example, the assembler recognizes ABC, Abc, and abc as three unique symbols. You can override case sensitivity with the `--syms_ignore_case` assembler option (see [Section 3.3](#)). A symbol is valid only during the assembly in which it is defined, unless you use the `.global` directive or the `.def` directive to declare it as an external symbol (see [Identify Global Symbols](#)).

3.8.1 Labels

Symbols used as labels become symbolic addresses that are associated with locations in the program. Labels used locally within a file must be unique. Mnemonic opcodes and assembler directive names without the `.` prefix are valid label names.

Labels can also be used as the operands of `.global`, `.ref`, `.def`, or `.bss` directives; for example:

```
.global func

MOV    #CON1, R11
MOV    R11, 0(SP)
CALL   #func
```

3.8.2 Local Labels

Local labels are special labels whose scope and effect are temporary. A local label can be defined in two ways:

- `$n`, where `n` is a decimal digit in the range 0-9. For example, `$4` and `$1` are valid local labels. See [Example 3-1](#).
- `name?`, where `name` is any legal symbol name as described above. The assembler replaces the question mark with a period followed by a unique number. When the source code is expanded, *you will not see the unique number in the listing file*. Your label appears with the question mark as it did in the source definition. You cannot declare this label as global.

Normal labels must be unique (they can be declared only once), and they can be used as constants in the operand field. Local labels, however, can be undefined and defined again. Local labels cannot be defined by directives.

A local label can be undefined or reset in one of these ways:

- By using the `.newblock` directive
- By changing sections (using a `.sect`, `.text`, or `.data` directive)
- By entering an include file (specified by the `.include` or `.copy` directive)
- By leaving an include file (specified by the `.include` or `.copy` directive)

Example 3-1. Local Labels of the Form \$n

This is an example of code that declares and uses a local label legally:

```
.global ADDRA, ADDRb, ADDRc
Label1: MOV #ADDRA, R11      ; Load Address A to R11.
        SUB #ADDRb, R11    ; Subtract Address B.
        JL $1              ; If < 0, branch to $1
        MOV #ADDRb, R11    ; otherwise, load ADDRb to R11
        JMP $2             ; and branch to $2.
$1      MOV #ADDRA, R11    ; $1: load ADDRA to AC0.
$2      ADD #ADDRc, R11    ; $2: add ADDRc.
        .newblock         ; Undefine $1 so it can be used again.
        JMP $1            ; If less than zero, branch to $1.
        MOV R11, &ADDRc   ; Store AC0 low in ADDRc.
$1 NOP
```

The following code uses a local label illegally:

```
.global ADDRA, ADDRb, ADDRc
Label1: MOV #ADDRA, R11    ; Load Address A to R11.
        SUB #ADDRb, R11    ; Subtract Address B.
        JL $1              ; If < 0, branch to $1
        MOV #ADDRb, R11    ; otherwise, load ADDRb to R11
        JMP $2             ; and branch to $2.
$1      MOV #ADDRA, R11    ; $1: load ADDRA to AC0.
$2      ADD #ADDRc, R11    ; $2: add ADDRc.
        JMP $1            ; If less than zero, branch to $1.
        MOV R11, &ADDRc   ; Store AC0 low in ADDRc.
$1 NOP
```

The \$1 label is not undefined before being reused by the second branch instruction. Therefore, \$1 is redefined, which is illegal.

Local labels are especially useful in macros. If a macro contains a normal label and is called more than once, the assembler issues a multiple-definition error. If you use a local label and `.newblock` within a macro, however, the local label is used and reset each time the macro is expanded.

Up to ten local labels of the \$n form can be in effect at one time. Local labels of the form name? are not limited. After you undefine a local label, you can define it and use it again. Local labels do not appear in the object code symbol table.

Because local labels are intended to be used only locally, branches to local labels are not expanded in case the branch's offset is out of range.

3.8.3 Symbolic Constants

Symbols can be set to constant values. By using constants, you can equate meaningful names with constant values. The `.set` and `.struct/.tag/.endstruct` directives enable you to set constants to symbolic names. Symbolic constants *cannot* be redefined. The following example shows how these directives can be used:

```
K      .set  1024           ; constant definitions
maxbuf .set  2*K

item  .struct           ; item structure definition
value .int              ; constant offsets value = 0
delta .int              ; constant offsets value = 1
i_len .endstruct

array .tag  item        ; array declaration
      .bss array, i_len*K
```

The assembler also has several predefined symbolic constants; these are discussed in [Section 3.8.5](#).

3.8.4 Defining Symbolic Constants (--asm_define Option)

The `--asm_define` option equates a constant value or a string with a symbol. The symbol can then be used in place of a value in assembly source. The format of the `--asm_define` option is as follows:

cl430 `--asm_define= name[=value]`

The *name* is the name of the symbol you want to define. The *value* is the constant or string value you want to assign to the symbol. If the *value* is omitted, the symbol is set to 1. If you want to define a quoted string and keep the quotation marks, do one of the following:

- For Windows, use `--asm_define=name="\value\"`. For example, `--asm_define=car="\sedan\"`
- For UNIX, use `--asm_define=name="value"`. For example, `--asm_define=car="sedan"`
- For Code Composer, enter the definition in a file and include that file with the `--cmd_file` (or `-@`) option.

Once you have defined the name with the `--asm_define` option, the symbol can be used in place of a constant value, a well-defined expression, or an otherwise undefined symbol used with assembly directives and instructions. For example, on the command line you enter:

```
cl430 --asm_define=SYM1=1 --asm_define=SYM2=2 --asm_define=SYM3=3 --asm_define=SYM4=4 value.asm
```

Since you have assigned values to SYM1, SYM2, SYM3, and SYM4, you can use them in source code. [Example 3-2](#) shows how the `value.asm` file uses these symbols without defining them explicitly.

Within assembler source, you can test the symbol defined with the `--asm_define` option with the following directives:

| Type of Test | Directive Usage |
|--------------------|--|
| Existence | <code>.if \$isdefed(" name ")</code> |
| Nonexistence | <code>.if \$isdefed(" name ") = 0</code> |
| Equal to value | <code>.if name = value</code> |
| Not equal to value | <code>.if name != value</code> |

The argument to the `$isdefed` built-in function must be enclosed in quotes. The quotes cause the argument to be interpreted literally rather than as a substitution symbol.

Example 3-2. Using Symbolic Constants Defined on Command Line

```

If_4:  .if      SYM4 = SYM2 * SYM2
        .byte   SYM4          ; Equal values
        .else
        .byte   SYM2 * SYM2  ; Unequal values
        .endif

IF_5:  .if      SYM1 <= 10
        .byte   10          ; Less than / equal
        .else
        .byte   SYM1        ; Greater than
        .endif

IF_6:  .if      SYM3 * SYM2 != SYM4 + SYM2
        .byte   SYM3 * SYM2 ; Unequal value
        .else
        .byte   SYM4 + SYM4 ; Equal values
        .endif

IF_7:  .if      SYM1 = SYM2
        .byte   SYM1
        .elseif SYM2 + SYM3 = 5
        .byte   SYM2 + SYM3
        .endif

```

3.8.5 Predefined Symbolic Constants

The assembler has several predefined symbols, including the following types:

- **\$**, the dollar-sign character, represents the current value of the section program counter (SPC). \$ is a relocatable symbol.
- **Register symbols** (the name of MSP430 registers) include R0-R15 and their aliases. The MSP430 register aliases are defined as follows:

| Register Name | Alias |
|---------------|-------|
| R0 | PC |
| R1 | SP |
| R2 | SR |

Register symbols and aliases can be entered as all uppercase or all lowercase characters; that is, R1 could also be entered as r1, SP, or sp.

- **Processor symbols**, as follow:
 - The `.MSP430` symbol is always set to 1. The processor symbol can be entered as all uppercase or all lowercase characters; that is, `.MSP430` could also be entered as `.msp430`.
 - The `.MSP4619` symbol is set to 1 when the `--silicon_version=msp4619` option is specified. The processor symbol can be entered as all uppercase or all lowercase characters; that is, `.MSP4619` could also be entered as `.msp4619`.
 - The `__LARGE_CODE_MODEL__` symbol is set to 1 when the `--version=msp4619` option is specified.
 - The `__LARGE_DATA_MODEL__` symbol is set to 1 when the `--version=msp4619` and `--large_memory_model` options are used.

3.8.6 Substitution Symbols

Symbols can be assigned a string value (variable). This enables you to alias character strings by equating them to symbolic names. Symbols that represent character strings are called substitution symbols. When the assembler encounters a substitution symbol, its string value is substituted for the symbol name. Unlike symbolic constants, substitution symbols can be redefined.

A string can be assigned to a substitution symbol anywhere within a program; for example:

```
.asg  "SP", stack-pointer
    ; Assigns the string SP to the substitution symbol
    ; stack-pointer.
.asg  "#0x20", block2
    ; Assigns the string #0x20 to the substitution
    ; symbol block2.
ADD   block2, stack-pointer,
    ; Adds the value in SP to #0x20 and stores the
    ; result in SP.
```

When you are using macros, substitution symbols are important because macro parameters are actually substitution symbols that are assigned a macro argument. The following code shows how substitution symbols are used in macros:

```
myadd .macro src, dest
    ; addl macro definition
    ADD src, dest
    ; Add the value in register dest to the value in
    ; register src.
.endm

*myadd invocation
myadd R4, R5
    ; Calls the macro addl and substitutes R4 for src
    ; and R5 for dest. The macro adds the value of R4
    ; and the value of R5.
```

See [Chapter 5](#) for more information about macros.

3.9 Expressions

An expression is a constant, a symbol, or a series of constants and symbols separated by arithmetic operators. The 32-bit ranges of valid expression values are -2147 483 648 to 2147 483 647 for signed values, and 0 to 4 294 967 295 for unsigned values. Three main factors influence the order of expression evaluation:

| | |
|---------------------------------|---|
| Parentheses | Expressions enclosed in parentheses are always evaluated first. $8 / (4 / 2) = 4$, but $8 / 4 / 2 = 1$ You <i>cannot</i> substitute braces ({ }) or brackets ([]) for parentheses. |
| Precedence groups | Operators, listed in Table 3-2 , are divided into nine precedence groups. When parentheses do not determine the order of expression evaluation, the highest precedence operation is evaluated first. $8 + 4 / 2 = 10$ (4 / 2 is evaluated first) |
| Left-to-right evaluation | When parentheses and precedence groups do not determine the order of expression evaluation, the expressions are evaluated from left to right, except for Group 1, which is evaluated from right to left. $8 / 4 * 2 = 4$, but $8 / (4 * 2) = 1$ |

3.9.1 Operators

Table 3-2 lists the operators that can be used in expressions, according to precedence group.

Table 3-2. Operators Used in Expressions (Precedence)

| Group ⁽¹⁾ | Operator | Description ⁽²⁾ |
|----------------------|----------|----------------------------|
| 1 | + | Unary plus |
| | - | Unary minus |
| | ~ | 1s complement |
| | ! | Logical NOT |
| 2 | * | Multiplication |
| | / | Division |
| | % | Modulo |
| 3 | + | Addition |
| | - | Subtraction |
| 4 | << | Shift left |
| | >> | Shift right |
| 5 | < | Less than |
| | <= | Less than or equal to |
| | > | Greater than |
| | >= | Greater than or equal to |
| 6 | =[=] | Equal to |
| | != | Not equal to |
| 7 | & | Bitwise AND |
| 8 | ^ | Bitwise exclusive OR (XOR) |
| 9 | | Bitwise OR |

⁽¹⁾ Group 1 operators are evaluated right to left. All other operators are evaluated left to right.

⁽²⁾ Unary + and - have higher precedence than the binary forms.

3.9.2 Expression Overflow and Underflow

The assembler checks for overflow and underflow conditions when arithmetic operations are performed at assembly time. It issues a warning (the message *Value Truncated*) whenever an overflow or underflow occurs. The assembler *does not* check for overflow or underflow in multiplication.

3.9.3 Well-Defined Expressions

Some assembler directives require well-defined expressions as operands. Well-defined expressions contain only symbols or assembly-time constants that are defined before they are encountered in the expression. The evaluation of a well-defined expression must be absolute.

This is an example of a well-defined expression:

```
1000h+X
```

where X was previously defined as an absolute symbol.

3.9.4 Conditional Expressions

The assembler supports relational operators that can be used in any expression; they are especially useful for conditional assembly. Relational operators include the following:

| | | | |
|---|--------------|----|--------------------------|
| = | Equal to | != | Not equal to |
| < | Less than | <= | Less than or equal to |
| > | Greater than | >= | Greater than or equal to |

Conditional expressions evaluate to 1 if true and 0 if false and can be used only on operands of equivalent types; for example, absolute value compared to absolute value, but not absolute value compared to relocatable value.

3.10 Source Listings

A source listing shows source statements and the object code they produce. To obtain a listing file, invoke the assembler with the `--asm_listing` option (see [Section 3.3](#)).

Two banner lines, a blank line, and a title line are at the top of each source listing page. Any title supplied by the `.title` directive is printed on the title line. A page number is printed to the right of the title. If you do not use the `.title` directive, the name of the source file is printed. The assembler inserts a blank line below the title line.

Each line in the source file produces at least one line in the listing file. This line shows a source statement number, an SPC value, the object code assembled, and the source statement. and show these in actual listing files.

Each line in the source file produces at least one line in the listing file. This line shows a source statement number, an SPC value, the object code assembled, and the source statement. [Figure 3-2](#) shows these in an actual listing file.

Field 1: Source Statement Number

Line number

The source statement number is a decimal number. The assembler numbers source lines as it encounters them in the source file; some statements increment the line counter but are not listed. (For example, `.title` statements and statements following a `.nolist` are not listed.) The difference between two consecutive source line numbers indicates the number of intervening statements in the source file that are not listed.

Include file letter

A letter preceding the line number indicates the line is assembled from the include file designated by the letter.

Nesting level number

A number preceding the line number indicates the nesting level of macro expansions or loop blocks.

Field 2: Section Program Counter

This field contains the SPC value, which is hexadecimal. All sections (`.text`, `.data`, `.bss`, and named sections) maintain separate SPCs. Some directives do not affect the SPC and leave this field blank.

Field 3: Object Code

This field contains the hexadecimal representation of the object code. All machine instructions and directives use this field to list object code. This field also indicates the relocation type associated with an operand for this line of source code. If more than one operand is relocatable, this column indicates the relocation type for the first operand. The characters that can appear in this column and their associated relocation types are listed below:

| | |
|---|---|
| ! | undefined external reference |
| ' | <code>.text</code> relocatable |
| + | <code>.sect</code> relocatable |
| " | <code>.data</code> relocatable |
| - | <code>.bss</code> , <code>.usect</code> relocatable |
| % | relocation expression |

Field 4: Source Statement Field

This field contains the characters of the source statement as they were scanned by the assembler. The assembler accepts a maximum line length of 200 characters. Spacing in this field is determined by the spacing in the source statement.

Figure 3-2 shows an assembler listing with each of the four fields identified.

Figure 3-2. Example Assembler Listing

| Include file letter | Line number | | |
|---------------------|-------------|------------|------------------|
| | 1 | | .copy "mac1.inc" |
| A | 1 | addfive | .macro dst |
| A | 2 | | ADD.W #5,dst |
| A | 3 | | .endm |
| | 2 | | |
| | 3 | | .global var1 |
| | 4 | | |
| | 5 | | |
| | 6 | 0000 430B | MOV.W #0,R11 |
| | 7 | | |
| | 8 | | .loop 5 |
| | 9 | | addfive R11 |
| | 10 | | .endloop |
| 1 | | 0002 | addfive R11 |
| 2 | | 0002 503B | ADD.W #5,R11 |
| | | 0004 0005 | |
| 1 | | 0006 | addfive R11 |
| 2 | | 0006 503B | ADD.W #5,R11 |
| | | 0008 0005 | |
| 1 | | 000a | addfive R11 |
| 2 | | 000a 503B | ADD.W #5,R11 |
| | | 000c 0005 | |
| 1 | | 000e | addfive R11 |
| 2 | | 000e 503B | ADD.W #5,R11 |
| | | 0010 0005 | |
| 1 | | 0012 | addfive R11 |
| 2 | | 0012 503B | ADD.W #5,R11 |
| | | 0014 0005 | |
| | 11 | | |
| | 12 | 0016 4B82 | MOV.W R11,&var1 |
| | | 0018 0000! | |

| | | | |
|---------|---------|---------|---------|
| Field 1 | Field 2 | Field 2 | Field 4 |
|---------|---------|---------|---------|

3.11 Debugging Assembly Source

When you invoke cl430 with `--symdebug:dwarf` (or `-g`) when compiling an assembly file, the assembler provides symbolic debugging information that allows you to step through your assembly code in a debugger rather than using the Disassembly window in Code Composer Studio. This enables you to view source comments and other source-code annotations while debugging.

The `.asmfunc` and `.endasmfunc` (see [Mark Function Boundaries](#)) directives enable you to use C characteristics in assembly code that makes the process of debugging an assembly file more closely resemble debugging a C/C++ source file.

The `.asmfunc` and `.endasmfunc` directives allow you to name certain areas of your code, and make these areas appear in the debugger as C functions. Contiguous sections of assembly code that are not enclosed by the `.asmfunc` and `.endasmfunc` directives are automatically placed in assembler-defined functions named with this syntax:

`$ filename : starting source line : ending source line $`

If you want to view your variables as a user-defined type in C code, the types must be declared and the variables must be defined in a C file. This C file can then be referenced in assembly code using the `.ref` directive (see [Identify Global Symbols](#)).

Example 3-3. Viewing Assembly Variables as C Types C Program

```
typedef struct
{
    int m1;
    int m2;
} X;

X svar = { 1, 2 };
```

Example 3-4. Assembly Program for [Example 3-3](#)

```
.ref svar

.global addfive

addfive: .asmfunc
MOV #5,R12
ADD R12,&svar
ADD R12,&svar + 2
RET
.endasmfunc
```

[Example 3-3](#) shows the `cvar.c` C program that defines a variable, `svar`, as the structure type `X`. The `svar` variable is then referenced in the `addfive.asm` assembly program in [Example 3-4](#) and 5 is added to `svar`'s second data member.

Compile both source files with the `--symdebug:dwarf` option (`-g`) and link them as follows:

```
cl430 --symdebug:dwarf cvars.c addfive.asm --run_linker --library=lnk.cmd --library=rts430.lib
--output_file=addfive.out
```

When you load this program into a symbolic debugger, `addfive` appears as a C function. You can monitor the values in `svar` while stepping through `main` just as you would any regular C variable.

3.12 Cross-Reference Listings

A cross-reference listing shows symbols and their definitions. To obtain a cross-reference listing, invoke the assembler with the `--cross_reference` option (see [Section 3.3](#)) or use the `.option` directive with the X operand (see [Select Listing Options](#)). The assembler appends the cross-reference to the end of the source listing. [Example 3-5](#) shows the four fields contained in the cross-reference listing.

Example 3-5. An Assembler Cross-Reference Listing

| LABEL | VALUE | DEFN | REF | |
|---------|-------|------|-----|---|
| .MSP430 | 0001 | 0 | | |
| .msp430 | 0001 | 0 | | |
| addfive | 0000' | 6 | 3 | |
| svar | REF | 1 | 7 | 8 |

- Label** column contains each symbol that was defined or referenced during the assembly.
- Value** column contains an 8-digit hexadecimal number (which is the value assigned to the symbol) or a name that describes the symbol's attributes. A value may also be preceded by a character that describes the symbol's attributes. [Table 3-3](#) lists these characters and names.
- Definition** (DEFN) column contains the statement number that defines the symbol. This column is blank for undefined symbols.
- Reference** (REF) column lists the line numbers of statements that reference the symbol. A blank in this column indicates that the symbol was never used.

Table 3-3. Symbol Attributes

| Character or Name | Meaning |
|-------------------|--|
| REF | External reference (global symbol) |
| UNDF | Undefined |
| ' | Symbol defined in a .text section |
| " | Symbol defined in a .data section |
| + | Symbol defined in a .sect section |
| - | Symbol defined in a .bss or .usect section |

Assembler Directives

Assembler directives supply data to the program and control the assembly process. Assembler directives enable you to do the following:

- Assemble code and data into specified sections
- Reserve space in memory for uninitialized variables
- Control the appearance of listings
- Initialize memory
- Assemble conditional blocks
- Define global variables
- Specify libraries from which the assembler can obtain macros
- Examine symbolic debugging information

This chapter is divided into two parts: the first part ([Section 4.1](#) through [Section 4.10](#)) describes the directives according to function, and the second part ([Section 4.11](#)) is an alphabetical reference.

| Topic | Page |
|--|-----------|
| 4.1 Directives Summary | 58 |
| 4.2 Directives That Define Sections | 61 |
| 4.3 Directives That Initialize Constants | 63 |
| 4.4 Directives That Perform Alignment and Reserve Space | 65 |
| 4.5 Directives That Format the Output Listings | 66 |
| 4.6 Directives That Reference Other Files | 67 |
| 4.7 Directives That Enable Conditional Assembly | 67 |
| 4.8 Directives That Define Structures..... | 68 |
| 4.9 Directives That Define Symbols at Assembly Time..... | 68 |
| 4.10 Miscellaneous Directives | 69 |
| 4.11 Directives Reference | 70 |

4.1 Directives Summary

Table 4-1 through Table 4-9 summarize the assembler directives.

Besides the assembler directives documented here, the MSP430™ software tools support the following directives:

- The assembler uses several directives for macros. Macro directives are discussed in [Chapter 5](#); they are not discussed in this chapter.
- The C compiler uses directives for symbolic debugging. Unlike other directives, symbolic debugging directives are not used in most assembly language programs. [Appendix A](#) discusses these directives; they are not discussed in this chapter.

Labels and Comments Are Not Shown in Syntaxes

Note: Any source statement that contains a directive can also contain a label and a comment. Labels begin in the first column (only labels and comments can appear in the first column), and comments must be preceded by a semicolon, or an asterisk if the comment is the only element in the line. To improve readability, labels and comments are not shown as part of the directive syntax.

Table 4-1. Directives That Define Sections

| Mnemonic and Syntax | Description | See |
|---|---|------------------------------|
| .bss <i>symbol, size in bytes[, alignment]</i> | Reserves <i>size</i> bytes in the .bss (uninitialized data) section | .bss topic |
| .data | Assembles into the .data (initialized data) section | .data topic |
| .sect " <i>section name</i> " | Assembles into a named (initialized) section | .sect topic |
| .text | Assembles into the .text (executable code) section | .text topic |
| symbol .usect " <i>section name</i> ", <i>size in bytes</i> [, <i>alignment</i>] | Reserves <i>size</i> bytes in a named (uninitialized) section | .usect topic |

Table 4-2. Directives That Initialize Constants (Data and Memory)

| Mnemonic and Syntax | Description | See |
|---|---|-------------------------------|
| .byte <i>value</i> ₁ [, ..., <i>value</i> _{<i>n</i>}] | Initializes one or more successive bytes in the current section | .byte topic |
| .char <i>value</i> ₁ [, ..., <i>value</i> _{<i>n</i>}] | Initializes one or more successive bytes in the current section | .char topic |
| .double <i>value</i> ₁ [, ..., <i>value</i> _{<i>n</i>}] | Initializes one or more 32-bit, IEEE double-precision, floating-point constants | .double topic |
| .field <i>value</i> [, <i>size</i>] | Initializes a field of <i>size</i> bits (1-32) with <i>value</i> | .field topic |
| .float <i>value</i> ₁ [, ..., <i>value</i> _{<i>n</i>}] | Initializes one or more 32-bit, IEEE single-precision, floating-point constants | .float topic |
| .half <i>value</i> ₁ [, ... , <i>value</i> _{<i>n</i>}] | Initializes one or more 16-bit integers (halfword) | .half topic |
| .int <i>value</i> ₁ [, ... , <i>value</i> _{<i>n</i>}] | Initializes one or more 16-bit integers | .int topic |
| .long <i>value</i> ₁ [, ... , <i>value</i> _{<i>n</i>}] | Initializes one or more 32-bit integers | .long topic |
| .short <i>value</i> ₁ [, ... , <i>value</i> _{<i>n</i>}] | Initializes one or more 16-bit integers (halfword) | .short topic |
| .string { <i>expr</i> ₁ " <i>string</i> ₁ "}[, ... , { <i>expr</i> _{<i>n</i>} " <i>string</i> _{<i>n</i>} "}] | Initializes one or more text strings | .string topic |
| .word <i>value</i> ₁ [, ... , <i>value</i> _{<i>n</i>}] | Initializes one or more 16-bit integers | .word topic |

Table 4-3. Directives That Perform Alignment and Reserve Space

| Mnemonic and Syntax | Description | See |
|--|---|------------------------------|
| .align [<i>size in bytes</i>] | Aligns the SPC on a boundary specified by <i>size in bytes</i> , which must be a power of 2; defaults to word (2-byte) boundary | .align topic |
| .bes <i>size</i> | Reserves <i>size</i> bytes in the current section; a label points to the end of the reserved space | .bes topic |
| .space <i>size</i> | Reserves <i>size</i> bytes in the current section; a label points to the beginning of the reserved space | .space topic |

Table 4-4. Directives That Format the Output Listing

| Mnemonic and Syntax | Description | See |
|--|--|---------------------------------|
| .drlist | Enables listing of all directive lines (default) | .drlist topic |
| .drnolist | Suppresses listing of certain directive lines | .drnolist topic |
| .fclist | Allows false conditional code block listing (default) | .fclist topic |
| .fcnolist | Suppresses false conditional code block listing | .fcnolist topic |
| .length [<i>page length</i>] | Sets the page length of the source listing | .length topic |
| .list | Restarts the source listing | .list topic |
| .mlist | Allows macro listings and loop blocks (default) | .mlist topic |
| .mnolist | Suppresses macro listings and loop blocks | .mnolist topic |
| .nolist | Stops the source listing | .nolist topic |
| .option <i>option₁</i> [, <i>option₂</i> , . . .] | Selects output listing options; available options are A, B, H, M, N, O, R, T, W, and X | .option topic |
| .page | Ejects a page in the source listing | .page topic |
| .sslist | Allows expanded substitution symbol listing | .sslist topic |
| .ssnolist | Suppresses expanded substitution symbol listing (default) | .ssnolist topic |
| .tab <i>size</i> | Sets tab to <i>size</i> characters | .tab topic |
| .title " <i>string</i> " | Prints a title in the listing page heading | .title topic |
| .width [<i>page width</i>] | Sets the page width of the source listing | .width topic |

Table 4-5. Directives That Reference Other Files

| Mnemonic and Syntax | Description | See |
|---|---|--------------------------------|
| .copy [" <i>filename</i> "] | Includes source statements from another file | .copy topic |
| .def <i>symbol₁</i> [, ... , <i>symbol_n</i>] | Identifies one or more symbols that are defined in the current module and that can be used in other modules | .def topic |
| .global <i>symbol₁</i> [, ... , <i>symbol_n</i>] | Identifies one or more global (external) symbols | .global topic |
| .include [" <i>filename</i> "] | Includes source statements from another file | .include topic |
| .mlib [" <i>filename</i> "] | Defines macro library | .mlib topic |
| .ref <i>symbol₁</i> [, ... , <i>symbol_n</i>] | Identifies one or more symbols used in the current module that are defined in another module | .ref topic |

Table 4-6. Directives That Enable Conditional Assembly

| Mnemonic and Syntax | Description | See |
|--|---|--------------------------------|
| .break [<i>well-defined expression</i>] | Ends .loop assembly if <i>well-defined expression</i> is true. When using the .loop construct, the .break construct is optional. | .break topic |
| .else | Assembles code block if the .if <i>well-defined expression</i> is false. When using the .if construct, the .else construct is optional. | .else topic |
| .elseif <i>well-defined expression</i> | Assembles code block if the .if <i>well-defined expression</i> is false and the .elseif condition is true. When using the .if construct, the .elseif construct is optional. | .elseif topic |
| .endif | Ends .if code block | .endif topic |
| .endloop | Ends .loop code block | .endloop topic |
| .if <i>well-defined expression</i> | Assembles code block if the <i>well-defined expression</i> is true | .if topic |
| .loop [<i>well-defined expression</i>] | Begins repeatable assembly of a code block; the loop count is determined by the <i>well-defined expression</i> . | .loop topic |

Table 4-7. Directives That Define Structures

| Mnemonic and Syntax | Description | See |
|---------------------|---|-------------------------------|
| .endstruct | Ends a structure definition | .struct topic |
| .struct | Begins structure definition | .struct topic |
| .tag | Assigns structure attributes to a label | .struct |

Table 4-8. Directives That Define Symbols at Assembly Time

| Mnemonic and Syntax | Description | See |
|--|--|------------------------------|
| .asg [" <i>character string</i> "], <i>substitution symbol</i> | Assigns a character string to <i>substitution symbol</i> | .asg topic |
| <i>symbol</i> .equ <i>value</i> | Equates <i>value</i> with <i>symbol</i> | .equ topic |
| .eval <i>well-defined expression</i> , <i>substitution symbol</i> | Performs arithmetic on a numeric <i>substitution symbol</i> | .eval topic |
| .label <i>symbol</i> | Defines a load-time relocatable label in a section | .label topic |
| <i>symbol</i> .set <i>value</i> | Equates <i>value</i> with <i>symbol</i> | .set topic |
| .var | Adds a local substitution symbol to a macro's parameter list | .var topic |

Table 4-9. Directives That Perform Miscellaneous Functions

| Mnemonic and Syntax | Description | See |
|--|---|-----------------------------------|
| .asmfunc | Identifies the beginning of a block of code that contains a function | .asmfunc topic |
| .cdecls [<i>options</i>], " <i>filename</i> ", " <i>filename2</i> ", ...] | Share C headers between C and assembly code | .cdecls topic |
| .clink [" <i>section name</i> "] | Enables conditional linking for the current or specified section | .clink topic |
| .emsg <i>string</i> | Sends user-defined error messages to the output device; produces no .obj file | .emsg topic |
| .end | Ends program | .end topic |
| .endasmfunc | Identifies the end of a block of code that contains a function | .endasmfunc topic |
| .mmsg <i>string</i> | Sends user-defined messages to the output device | .mmsg topic |
| .newblock | Undefines local labels | .newblock topic |
| .wmsg <i>string</i> | Sends user-defined warning messages to the output device | .wmsg topic |

4.2 Directives That Define Sections

These directives associate portions of an assembly language program with the appropriate sections:

- The **.bss** directive reserves space in the .bss section for uninitialized variables.
- The **.data** directive identifies portions of code in the .data section. The .data section usually contains initialized data.
- The **.sect** directive defines an initialized named section and associates subsequent code or data with that section. A section defined with .sect can contain code or data.
- The **.text** directive identifies portions of code in the .text section. The .text section usually contains executable code.
- The **.usect** directive reserves space in an uninitialized named section. The .usect directive is similar to the .bss directive, but it allows you to reserve space separately from the .bss section.

[Chapter 2](#) discusses these sections in detail.

[Example 4-1](#) shows how you can use sections directives to associate code and data with the proper sections. This is an output listing; column 1 shows line numbers, and column 2 shows the SPC values. (Each section has its own program counter, or SPC.) When code is first placed in a section, its SPC equals 0. When you resume assembling into a section after other code is assembled, the section's SPC resumes counting as if there had been no intervening code.

The directives in [Example 4-1](#) perform the following tasks:

| | |
|-----------------|--|
| .text | initializes words with the values 1, 2, 3, 4, 5, 6, 7, and 8. |
| .data | initializes words with the values 9, 10, 11, 12, 13, 14, 15, and 16. |
| var_defs | initializes words with the values 17 and 18. |
| .bss | reserves 19 bytes. |
| xy | reserves 20 bytes. |

The .bss and .usect directives do not end the current section or begin new sections; they reserve the specified amount of space, and then the assembler resumes assembling code or data into the current section.

Example 4-1. Sections Directives

```

1          ; Comment1 here : Start assembling ....
2
3 0000          .text
4 0000 0001          .word   1,2
   0002 0002
5 0004 0003          .word   3,4
   0006 0004
6
7          ; Comment 2
8
9 0000          .data
10 0000 0009          .word   9,10
   0002 000A
11 0004 000B          .word   11,12
   0006 000C
12
13          ; Comment 3
14
15 0000          .sect    "var_defs"
16 0000 0011          .word   17,18
   0002 0012
17
18          ; Comment 4
19
20 0008          .data
21 0008 000D          .word   13,14
   000a 000E
22
23 0000          .bss     sym,19
24 000c 000F          .word   15,16
   000e 0010
25
26          ; Comment 5
27
28 0008          .text
29 0008 0005          .word   5,6
   000a 0006
30 0000 usym          .usect  "xy",20
31 000c 0007          .word   7,8
   000e 0008

```

4.3 Directives That Initialize Constants

Several directives assemble values for the current section:

- The **.byte** and **.char** directives place one or more 8-bit values into consecutive bytes of the current section. These directives are similar to **.long** and **.word**, except that the width of each value is restricted to eight bits.
- The **.double** and **.float** directives calculate the single-precision (32-bit) IEEE floating-point representation of a single floating-point value and store it in a word in the current section that is aligned to a word boundary.
- The **.field** directive places a single value into a specified number of bits in the current word. With **.field**, you can pack multiple fields into a single word; the assembler does not increment the SPC until a word is filled.

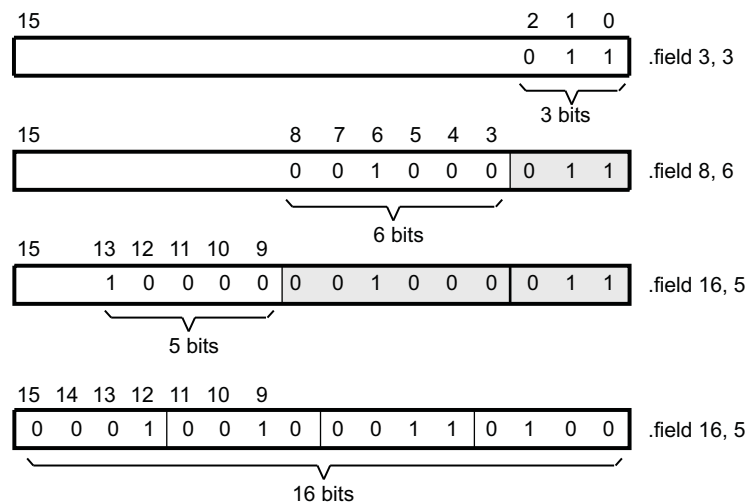
Figure 4-1 shows how fields are packed into a word. Using the following assembled code, notice that the SPC does not change for the first three fields (the fields are packed into the same word):

```

1 0000 0003      .field 3,3
2 0000 0043      .field 8,6
3 0000 2043      .field 16,5
4 0002 1234      .field 0x1234,16

```

Figure 4-1. The **.field** Directive



- The **.int** and **.word** directives place one or more 16-bit values into consecutive 16-bit values in the current section.
- The **.string** directive places 8-bit characters from one or more character strings into the current section. The **.string** directive is similar to **.byte**, placing an 8-bit character in each consecutive byte of the current section.

Directives That Initialize Constants When Used in a **.struct/endstruct** Sequence

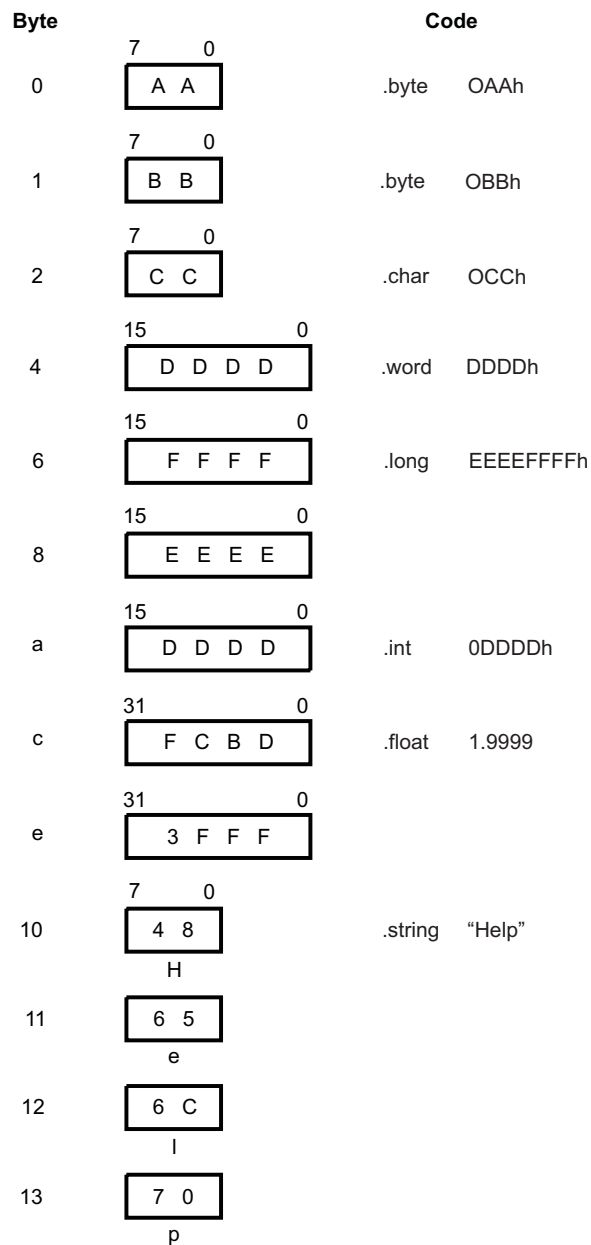
Note: The **.byte**, **.char**, **.word**, **.int**, **.long**, **.string**, **.double**, **.float**, **.half**, **.short**, and **.field** directives do not initialize memory when they are part of a **.struct/ .endstruct** sequence; rather, they define a member's size. For more information, see the [.struct/endstruct directives](#).

Figure 4-2 compares the .byte, .char, .int, .long, .float, .word, and .string directives. Using the following assembled code:

```

1 0000 00AA      .byte  0AAh,0BBh
   0001 00BB
2 0002 00CC      .char  0xCC
3 0004 DDDD      .word  0xDDDD
4 0006 FFFF      .long  0xFFFFFFFF
   0008 EEEE
5 000a DDDD      .int   0xDDDD
6 000c FCB9      .float 1.9999
   000e 3FFF
7 0010 0048      .string "Hello"
   0011 0065
   0012 006C
   0013 0070
    
```

Figure 4-2. Initialization Directives



4.4 Directives That Perform Alignment and Reserve Space

These directives align the section program counter (SPC) or reserve space in a section:

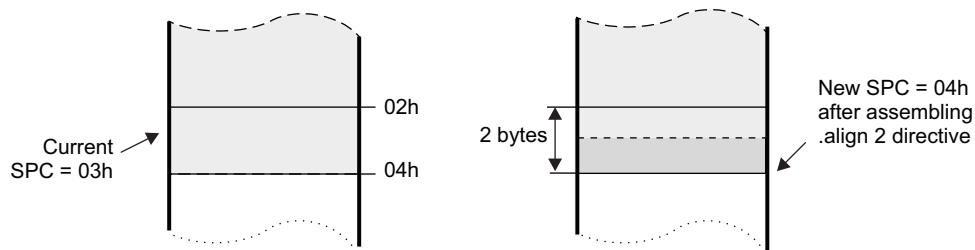
- The **.align** directive aligns the SPC at a 1-byte to 32K-byte boundary. This ensures that the code following the directive begins on the byte value that you specify. If the SPC is already aligned at the selected boundary, it is not incremented. Operands for the **.align** directive must equal a power of 2 between 2^0 and 2^{15} , inclusive.

Figure 4-3 demonstrates the **.align** directive. Using the following assembled code:

```

1 0000 0002      .field  2,3
2 0000 005A      .field  11,5
3                .align  2
4 0002 0045      .string "Err"
   0003 0072
   0004 0072
5                .align
6 0006 0004      .byte   4
  
```

Figure 4-3. The **.align** Directive



- The **.bes** and **.space** directives reserve a specified number of bytes in the current section. The assembler fills these reserved bytes with 0s.
 - When you use a label with **.space**, it points to the *first* byte that contains reserved bits.
 - When you use a label with **.bes**, it points to the *last* byte that contains reserved bits.

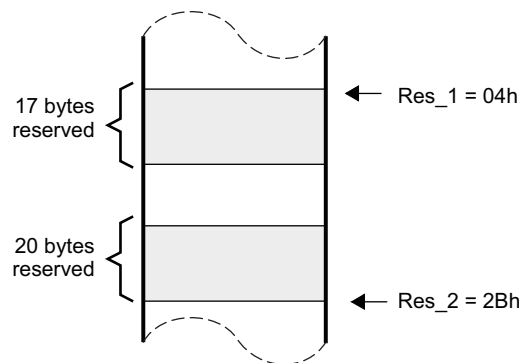
Figure 4-4 shows how the **.space** and **.bes** directives work for the following assembled code:

```

1 0000 0100      .word  0x100,0x200
   0002 0200
2 0004          Res_1  .space 17
3 0016 000F      .word  15
4 002b          Res_2  .bes   20
5 002c 00BA      .byte  0xBA
  
```

Res_1 points to the first byte in the space reserved by **.space**. Res_2 points to the last byte in the space reserved by **.bes**.

Figure 4-4. The **.space** and **.bes** Directives



4.5 Directives That Format the Output Listings

These directives format the listing file:

- The **.drlist** directive causes printing of the directive lines to the listing; the **.drnolist** directive turns it off for certain directives. You can use the **.drnolist** directive to suppress the printing of the following directives. You can use the **.drlist** directive to turn the listing on again.

| | | | | |
|---------------------|------------------------|----------------------|------------------------|---------------------|
| <code>.asg</code> | <code>.eval</code> | <code>.length</code> | <code>.mnolist</code> | <code>.var</code> |
| <code>.break</code> | <code>.fclist</code> | <code>.mlist</code> | <code>.sslist</code> | <code>.width</code> |
| <code>.emsg</code> | <code>.fcnolist</code> | <code>.mmsg</code> | <code>.ssnolist</code> | <code>.wmsg</code> |

- The source code listing includes false conditional blocks that do not generate code. The **.fclist** and **.fcnolist** directives turn this listing on and off. You can use the **.fclist** directive to list false conditional blocks exactly as they appear in the source code. You can use the **.fcnolist** directive to list only the conditional blocks that are actually assembled.
- The **.length** directive controls the page length of the listing file. You can use this directive to adjust listings for various output devices.
- The **.list** and **.nolist** directives turn the output listing on and off. You can use the **.nolist** directive to prevent the assembler from printing selected source statements in the listing file. Use the **.list** directive to turn the listing on again.
- The source code listing includes macro expansions and loop blocks. The **.mlist** and **.mnolist** directives turn this listing on and off. You can use the **.mlist** directive to print all macro expansions and loop blocks to the listing, and the **.mnolist** directive to suppress this listing.
- The **.option** directive controls certain features in the listing file. This directive has the following operands:
 - A** turns on listing of all directives and data, and subsequent expansions, macros, and blocks.
 - B** limits the listing of `.byte` and `.char` directives to one line.
 - H** limits the listing of `.half` and `.short` directives to one line.
 - M** turns off macro expansions in the listing.
 - N** turns off listing (performs `.nolist`).
 - O** turns on listing (performs `.list`).
 - R** resets the B, H, M, T, and, W directives (turns off the limits of B, H, M, T, and W).
 - T** limits the listing of `.string` directives to one line.
 - W** limits the listing of `.word` and `.int` directives to one line.
 - X** produces a cross-reference listing of symbols. You can also obtain a cross-reference listing by invoking the assembler with the `--cross_reference` option (see [Section 3.3](#)).
- The **.page** directive causes a page eject in the output listing.
- The source code listing includes substitution symbol expansions. The **.sslist** and **.ssnolist** directives turn this listing on and off. You can use the **.sslist** directive to print all substitution symbol expansions to the listing, and the **.ssnolist** directive to suppress this listing. These directives are useful for debugging the expansion of substitution symbols.
- The **.tab** directive defines tab size.
- The **.title** directive supplies a title that the assembler prints at the top of each page.
- The **.width** directive controls the page width of the listing file. You can use this directive to adjust listings for various output devices.

4.6 Directives That Reference Other Files

These directives supply information for or about other files that can be used in the assembly of the current file:

- The **.copy** and **.include** directives tell the assembler to begin reading source statements from another file. When the assembler finishes reading the source statements in the copy/include file, it resumes reading source statements from the current file. The statements read from a copied file are printed in the listing file; the statements read from an included file are *not* printed in the listing file.
- The **.def** directive identifies a symbol that is defined in the current module and that can be used in another module. The assembler includes the symbol in the symbol table.
- The **.global** directive declares a symbol external so that it is available to other modules at link time. (For more information about global symbols, see [Section 2.7.1](#)). The **.global** directive does double duty, acting as a **.def** for defined symbols and as a **.ref** for undefined symbols. The linker resolves an undefined global symbol reference only if the symbol is used in the program. The **.global** directive declares a 16-bit symbol.
- The **.mlib** directive supplies the assembler with the name of an archive library that contains macro definitions. When the assembler encounters a macro that is not defined in the current module, it searches for it in the macro library specified with **.mlib**.
- The **.ref** directive identifies a symbol that is used in the current module but is defined in another module. The assembler marks the symbol as an undefined external symbol and enters it in the object symbol table so the linker can resolve its definition. The **.ref** directive forces the linker to resolve a symbol reference.

4.7 Directives That Enable Conditional Assembly

Conditional assembly directives enable you to instruct the assembler to assemble certain sections of code according to a true or false evaluation of an expression. Two sets of directives allow you to assemble conditional blocks of code:

- The **.if/elseif/else/endif** directives tell the assembler to conditionally assemble a block of code according to the evaluation of an expression.

| | |
|--|---|
| .if <i>well-defined expression</i> | marks the beginning of a conditional block and assembles code if the <i>well-defined expression</i> is true. |
| [.elseif <i>well-defined expression</i>] | marks a block of code to be assembled if the <i>well-defined expression</i> is false and the .elseif condition is true. |
| .else | marks a block of code to be assembled if the <i>well-defined expression</i> is false and any .elseif conditions are false. |
| .endif | marks the end of a conditional block and terminates the block. |
- The **.loop/break/endloop** directives tell the assembler to repeatedly assemble a block of code according to the evaluation of an expression.

| | |
|--|---|
| .loop [<i>well-defined expression</i>] | marks the beginning of a repeatable block of code. The optional expression evaluates to the loop count. |
| .break [<i>well-defined expression</i>] | tells the assembler to assemble repeatedly when the <i>well-defined expression</i> is false and to go to the code immediately after .endloop when the expression is true or omitted. |
| .endloop | marks the end of a repeatable block. |

The assembler supports several relational operators that are useful for conditional expressions. For more information about relational operators, see [Section 3.9.4](#).

4.8 Directives That Define Structures

The **.struct/endstruct** directives set up C-like structure definitions. The **.union/endunion** directives set up C-like union definitions. The **.tag** directive assigns the C-like structure or union characteristics to a label.

The **.struct/endstruct** directives allow you to organize your information into structures so that similar elements can be grouped together. Similarly, the **.union/endunion** directives allow you to organize your information into unions. Element offset calculation is left up to the assembler. These directives do not allocate memory. They simply create a symbolic template that can be used repeatedly.

The **.tag** directive assigns a label to a structure or union. This simplifies the symbolic representation and also provides the ability to define structures that contain other structures. The **.tag** directive does not allocate memory, and the structure tag (**stag**) must be defined before it is used.

```

type .struct          ; structure tag definition
X   .int
Y   .int
T_LEN .endstruct

COORD .tag type      ; declare COORD (coordinate)

COORD .space T_LEN   ; actual memory allocation
LDR  R0, COORD.Y     ; load member Y of structure
                        ; COORD into register R0.
    
```

4.9 Directives That Define Symbols at Assembly Time

Assembly-time symbol directives equate meaningful symbol names to constant values or strings.

- The **.asg** directive assigns a character string to a substitution symbol. The value is stored in the substitution symbol table. When the assembler encounters a substitution symbol, it replaces the symbol with its character string value. Substitution symbols can be redefined.

```

.asg "10, 20, 30, 40", coefficients
    ; Assign string to substitution symbol.
.byte coefficients
    ; Place the symbol values 10, 20, 30, and 40
    ; into consecutive bytes in current section.
    
```

- The **.eval** directive evaluates a well-defined expression, translates the results into a character string, and assigns the character string to a substitution symbol. This directive is most useful for manipulating counters:

```

.asg 1, x ; x = 1
.loop ; Begin conditional loop.
.byte x*10h ; Store value into current section.
.break x = 4 ; Break loop if x = 4.
.eval x+1, x ; Increment x by 1.
.endloop ; End conditional loop.
    
```

- The **.label** directive defines a special symbol that refers to the load-time address within the current section. This is useful when a section loads at one address but runs at a different address. For example, you may want to load a block of performance-critical code into slower off-chip memory to save space and move the code to high-speed on-chip memory to run. See the [.label topic](#) for an example using a load-time address label.
- The **.set** and **.equ** directives set a constant value to a symbol. The symbol is stored in the symbol table and cannot be redefined; for example:

```

bval .set 0100h ; Set bval = 0100h
     .long bval, bval*2, bval+12
     ; Store the values 0100h, 0200h, and 010Ch
     ; into consecutive words in current section.
    
```

The **.set** and **.equ** directives produce no object code. The two directives are identical and can be used interchangeably.

- The **.var** directive allows you to use substitution symbols as local variables within a macro.

4.10 Miscellaneous Directives

These directives enable miscellaneous functions or features:

- The **.asmfunc** and **.endasmfunc** directives mark function boundaries. These directives are used with the compiler `--symdebug:dwarf (-g)` option to generate debug information for assembly functions.
- The **.cdecls** directive enables programmers in mixed assembly and C/C++ environments to share C headers containing declarations and prototypes between C and assembly code.
- The **.clink** directive enables conditional linking by telling the linker to leave the named section out of the final object module output of the linker if there are no references found to any symbol in the section. The **.clink** directive can be applied to initialized or uninitialized sections.
- The **.end** directive terminates assembly. If you use the **.end** directive, it should be the last source statement of a program. This directive has the same effect as an end-of-file character.
- The **.newblock** directive resets local labels. Local labels are symbols of the form `$n`, where `n` is a decimal digit, or of the form `NAME?`, where you specify `NAME`. They are defined when they appear in the label field. Local labels are temporary labels that can be used as operands for jump instructions. The **.newblock** directive limits the scope of local labels by resetting them after they are used. See [Section 3.8.2](#) for information on local labels.

These three directives enable you to define your own error and warning messages:

- The **.emsg** directive sends error messages to the standard output device. The **.emsg** directive generates errors in the same manner as the assembler, incrementing the error count and preventing the assembler from producing an object file.
- The **.mmsg** directive sends assembly-time messages to the standard output device. The **.mmsg** directive functions in the same manner as the **.emsg** and **.wmsg** directives but does not set the error count or the warning count. It does not affect the creation of the object file.
- The **.wmsg** directive sends warning messages to the standard output device. The **.wmsg** directive functions in the same manner as the **.emsg** directive but increments the warning count rather than the error count. It does not affect the creation of the object file.

For more information about using the error and warning directives in macros, see [Section 5.7](#).

4.11 Directives Reference

The remainder of this chapter is a reference. Generally, the directives are organized alphabetically, one directive per topic. Related directives (such as `.if/.else/.endif`), however, are presented together in one topic.

.align

Align SPC on the Next Boundary

Syntax

.align [*size in bytes*]

Description

The **.align** directive aligns the section program counter (SPC) on the next boundary, depending on the *size in bytes* parameter. The *size* can be any power of 2 between 2^0 and 2^{15} , inclusive. An operand of 2 aligns the SPC on the next word boundary, and this is the default if no size is given. For example:

```
2          aligns SPC to word boundary
4          aligns SPC to 2 word boundary
128       aligns SPC to 128-byte boundary
```

Using the `.align` directive has two effects:

- The assembler aligns the SPC on an x-byte boundary *within* the current section.
- The assembler sets a flag that forces the linker to align the section so that individual alignments remain intact when a section is loaded into memory.

Example

This example shows several types of alignment, including `.align 4`, `.align 8`, and a default `.align`.

```
1 0000 0004      .byte 4
2                .align 2
3 0002 0045      .string "Errorcnt"
  0003 0072
  0004 0072
  0005 006F
  0006 0072
  0007 0063
  0008 006E
  0009 0074
4                .align
5 000a 0003      .field 3,3
6 000a 002B      .field 5,4
7                .align 2
8 000c 0003      .field 3,3
9                .align 8
10 0010 0005     .field 5,4
11               .align
12 0012 0004     .byte 4
```

.asg/.eval
Assign a Substitution Symbol

Syntax

.asg ["*character string*"], *substitution symbol*

.eval *well-defined expression*, *substitution symbol*

Description

The **.asg** directive assigns character strings to substitution symbols. Substitution symbols are stored in the substitution symbol table. The **.asg** directive can be used in many of the same ways as the **.set** directive, but while **.set** assigns a constant value (which cannot be redefined) to a symbol, **.asg** assigns a character string (which can be redefined) to a substitution symbol.

- The assembler assigns the *character string* to the substitution symbol. The quotation marks are optional. If there are no quotation marks, the assembler reads characters up to the first comma and removes leading and trailing blanks. In either case, a character string is read and assigned to the substitution symbol.
- The *substitution symbol* must be a valid symbol name. The substitution symbol is up to 128 characters long and must begin with a letter. Remaining characters of the symbol can be a combination of alphanumeric characters, the underscore (`_`), and the dollar sign (`$`).

The **.eval** directive performs arithmetic on substitution symbols, which are stored in the substitution symbol table. This directive evaluates the *well-defined expression* and assigns the string value of the result to the substitution symbol. The **.eval** directive is especially useful as a counter in **.loop/.endloop** blocks.

- The *well-defined expression* is an alphanumeric expression in which all symbols have been previously defined in the current source module, so that the result is an absolute.
- The *substitution symbol* must be a valid symbol name. The substitution symbol is up to 128 characters long and must begin with a letter. Remaining characters of the symbol can be a combination of alphanumeric characters, the underscore (`_`), and the dollar sign (`$`).

Example

This example shows how .asg and .eval can be used.

```

1          .sslist          ; show expanded sub. symbols
2          ; using .asg and .eval
3
4          .asg      R12, LOCALSTACK
5          .asg      &, AND
6
7 0000 503C      ADD      #280 AND 255, LOCALSTACK
#          ADD      #280 & 255, R12
          0002 0018
8
9          .asg      0,x
10         .loop     5
11         .eval     x+1, x
12         .word     x
13         .endloop
1         .eval     x+1, x
#         .eval     0+1, x
1         0004 0001      .word     x
#         .word     1
1         .eval     x+1, x
#         .eval     1+1, x
1         0006 0002      .word     x
#         .word     2
1         .eval     x+1, x
#         .eval     2+1, x
1         0008 0003      .word     x
#         .word     3
1         .eval     x+1, x
#         .eval     3+1, x
1         000a 0004      .word     x
#         .word     4
1         .eval     x+1, x
#         .eval     4+1, x
1         000c 0005      .word     x
#         .word     5

```

.asmfunc/.endasmfunc *Mark Function Boundaries*

Syntax*symbol* **.asmfunc****.endasmfunc****Description**

The **.asmfunc** and **.endasmfunc** directives mark function boundaries. These directives are used with the compiler -g option (--symdebug:dwarf) to allow assembly code sections to be debugged in the same manner as C/C++ functions.

You should not use the same directives generated by the compiler (see [Appendix A](#)) to accomplish assembly debugging; those directives should be used only by the compiler to generate symbolic debugging information for C/C++ source files.

The **.asmfunc** and **.endasmfunc** directives cannot be used when invoking the compiler with the backwards-compatibility --symdebug:coff option. This option instructs the compiler to use the obsolete COFF symbolic debugging format, which does not support these directives.

The *symbol* is a label that must appear in the label field.

Consecutive ranges of assembly code that are not enclosed within a pair of **.asmfunc** and **.endasmfunc** directives are given a default name in the following format:

\$ filename : beginning source line : ending source line \$

.bss *Reserve Space in the .bss Section*
Syntax `.bss symbol, size in bytes[, alignment]`
Description The **.bss** directive reserves space for variables in the **.bss** section. This directive is usually used to allocate space in RAM.

- The *symbol* is a required parameter. It defines a label that points to the first location reserved by the directive. The symbol name must correspond to the variable that you are reserving space for.
- The *size in bytes* is a required parameter; it must be an absolute expression. The assembler allocates size bytes in the **.bss** section. There is no default size.
- The *alignment* is an optional parameter that ensures that the space allocated to the symbol occurs on the specified boundary. The boundary indicates the size of the alignment in bytes and must be set to a power of 2 between 2^0 and 2^{15} , inclusive. If the SPC is aligned at the specified boundary, it is not incremented.

For more information about sections, see [Chapter 2](#).

Example In this example, the **.bss** directive allocates space for two variables, TEMP and ARRAY. The symbol TEMP points to four bytes of uninitialized space (at **.bss** SPC = 0). The symbol ARRAY points to 100 bytes of uninitialized space (at **.bss** SPC = 04h). Symbols declared with the **.bss** directive can be referenced in the same manner as other symbols and can also be declared external.

```

1          *****
2          ** Start assembling into the .text section. **
3          *****
4 0000          .text
5 0000 430A      MOV     #0, R10
6
7          *****
8          ** Allocate 4 bytes in .bss for TEMP. **
9          *****
10 0000      Var_1: .bss    TEMP, 4
11
12          *****
13          ** Still in .text. **
14          *****
15 0002 503B      ADD     #56h, R11
   0004 0056
16 0006 5C0B      ADD     R12, R11
17
18          *****
19          ** Allocate 100 bytes in .bss for the symbol **
20          ** named ARRAY. **
21          *****
22 0004          .bss    ARRAY, 100, 4
23
24          *****
25          ** Assemble more code into .text. **
26          *****
27 0008 4130      RET
28
29          *****
30          ** Declare external .bss symbols. **
31          *****
32          .global ARRAY, TEMP
33          .end

```

.byte/.char
Initialize Byte

Syntax

```
.byte value1[, ... , valuen]
```

```
.char value1[, ... , valuen]
```

Description

The **.byte** and **.char** directives place one or more values into consecutive bytes of the current section. A *value* can be one of the following:

- An expression that the assembler evaluates and treats as an 8-bit signed number
- A character string enclosed in double quotes. Each character in a string represents a separate value, and values are stored in consecutive bytes. The entire string *must* be enclosed in quotes.

The assembler truncates values greater than eight bits. You can use up to 100 value parameters, but the total line length cannot exceed 200 characters.

If you use a label, it points to the location of the first byte that is initialized.

When you use these directives in a **.struct/.endstruct** sequence, they define a member's size; they do not initialize memory. For more information, see the [.struct/.endstruct/.tag topic](#).

Example

In this example, 8-bit values (10, -1, abc, and a) are placed into consecutive bytes in memory with **.byte**. Also, 8-bit values (8, -3, def, and b) are placed into consecutive bytes in memory with **.char**. The label STRX has the value 0h, which is the location of the first initialized byte. The label STRY has the value 6h, which is the first byte initialized by the **.char** directive.

```

1 0000                                .space 100h
2 0100 000A STRX                      .byte 10, -1, "abc", 'a'
   0101 00FF
   0102 0061
   0103 0062
   0104 0063
   0105 0061
3 0106 0008                          .char 8, -3, "def", 'b'
   0107 00FD
   0108 0064
   0109 0065
   010a 0066
   010b 0062

```

.cdecls
Share C Headers Between C and Assembly Code

Syntax
Single Line:

```
.cdecls [options,] "filename"[, "filename2"[,...]]
```

Syntax
Multiple Lines:

```
.cdecls [options]
```

```
{
```

```
/*-----*/
```

```
/* C/C++ code - Typically a list of #includes and a few defines */
```

```
/*-----*/
```

```
}
```

Description

The **.cdecls** directive allows programmers in mixed assembly and C/C++ environments to share C headers containing declarations and prototypes between the C and assembly code. Any legal C/C++ can be used in a **.cdecls** block and the C/C++ declarations cause suitable assembly to be generated automatically, allowing you to reference the C/C++ constructs in assembly code; such as calling functions, allocating space, and accessing structure members; using the equivalent assembly mechanisms. While function and variable definitions are ignored, most common C/C++ elements are converted to assembly, for instance: enumerations, (non-function-like) macros, function and variable prototypes, structures, and unions.

The **.cdecls** options control whether the code is treated as C or C++ code; and how the **.cdecls** block and converted code are presented. Options must be separated by commas; they can appear in any order:

- | | |
|---------------|---|
| C | Treat the code in the .cdecls block as C source code (default). |
| CPP | Treat the code in the .cdecls block as C++ source code. This is the opposite of the C option. |
| NOLIST | Do not include the converted assembly code in any listing file generated for the containing assembly file (default). |
| LIST | Include the converted assembly code in any listing file generated for the containing assembly file. This is the opposite of the NOLIST option. |
| NOWARN | Do not emit warnings on STDERR about C/C++ constructs that cannot be converted while parsing the .cdecls source block (default). |
| WARN | Generate warnings on STDERR about C/C++ constructs that cannot be converted while parsing the .cdecls source block. This is the opposite of the NOWARN option. |

In the single-line format, the options are followed by one or more filenames to include. The filenames and options are separated by commas. Each file listed acts as if **#include "filename"** was specified in the multiple-line format.

In the multiple-line format, the line following **.cdecls** must contain the opening **.cdecls** block indicator **{**. Everything after the **{**, up to the closing block indicator **}**, is treated as C/C++ source and processed. Ordinary assembler processing then resumes on the line following the closing **}**.

The text within **{** and **}** is passed to the C/C++ compiler to be converted into assembly language. Much of C language syntax, including function and variable definitions as well as function-like macros, is not supported and is ignored during the conversion. However, all of what traditionally appears in C header files is supported, including function and variable prototypes; structure and union declarations; non-function-like macros; enumerations; and **#define**'s.

The resulting assembly language is included in the assembly file at the point of the `.cdecls` directive. If the `LIST` option is used, the converted assembly statements are printed in the listing file.

The assembly resulting from the `.cdecls` directive is treated similarly to a `.include` file. Therefore the `.cdecls` directive can be nested within a file being copied or included. The assembler limits nesting to ten levels; the host operating system may set additional restrictions. The assembler precedes the line numbers of copied files with a letter code to identify the level of copying. An `A` indicates the first copied file, `B` indicates a second copied file, etc.

The `.cdecls` directive can appear anywhere in an assembly source file, and can occur multiple times within a file. However, the C/C++ environment created by one `.cdecls` is **not** inherited by a later `.cdecls`; the C/C++ environment starts new for each `.cdecls`.

See [Chapter 12](#) for more information on setting up and using the `.cdecls` directive with C header files.

Example

In this example, the `.cdecls` directive is used call the C header.h file.

C header file:

```
#define WANT_ID 10
#define NAME "John\n"

extern int a_variable;
extern float cvt_integer(int src);

struct myCstruct { int member_a; float member_b; };

enum status_enum { OK = 1, FAILED = 256, RUNNING = 0 };
```

Source file:

```
.cdecls C,LIST,"myheader.h"

size:      .int $sizeof(myCstruct)
aoffset:   .int myCstruct.member_a
boffset:   .int myCstruct.member_b
okvalue:   .int status_enum.OK
failval:   .int status_enum.FAILED
           .if $$defined(WANT_ID)
id         .cstring NAME
           .endif
```

Listing File:

```

1          .cdecls C,LIST,"myheader.h"
A 1          ; -----
A 2          ; Assembly Generated from C/C++ Source Code
A 3          ; -----
A 4
A 5          ; ===== MACRO DEFINITIONS =====
A 6          .define "1",WANT_ID
A 7          .define ""John\n"",NAME
A 8          .define "1",_OPTIMIZE_FOR_SPACE
A 9
A 10         ; ===== TYPE DEFINITIONS =====
A 11         status_enum      .enum
A 12         0001 OK          .emember 1
A 13         0100 FAILED     .emember 256
A 14         0000 RUNNING    .emember 0
A 15         .endenum
A 16
A 17         myCstruct        .struct 0,2          ; struct size=(6 bytes|48 bits), alignment=2
A 18         0000 member_a    .field 16         ; int member_a - offset 0 bytes, size (2
bytes|16 bits)
A 19         0002 member_b    .field 32         ; float member_b - offset 2 bytes, size (4
bytes|32 bits)
```

```

A 20      0006      .endstruct      ; final size=(6 bytes|48 bits)
A 21
A 22      ; ===== EXTERNAL FUNCTIONS =====
A 23      .global cvt_integer
A 24
A 25      ; ===== EXTERNAL VARIABLES =====
A 26      .global a_variable
      2
      3 0000 0006 size: .int $sizeof(myCstruct)
      4 0002 0000 aoffset: .int myCstruct.member_a
      5 0004 0002 boffset: .int myCstruct.member_b
      6 0006 0001 okvalue: .int status_enum.OK
      7 0008 0100 failval: .int status_enum.FAILED
      8      .if $defined(WANT_ID)
      9 000a 004A id      .cstring NAME
      000b 006F
      000c 0068
      000d 006E
      000e 000A
      000f 0000
10      .endif

```

.clink *Conditionally Leave Section Out of Object Module Output*

Syntax `.clink ["section name"]`

Description The **.clink** directive enables conditional linking by telling the linker to leave a section out of the final object module output of the linker if there are no references found to any symbol in *section name*. The **.clink** directive can be applied to initialized or uninitialized sections.

The *section name* identifies the section. If **.clink** is used without a section name, it applies to the current initialized section. If **.clink** is applied to an uninitialized section, the section name is required. The section name is significant to 200 characters and must be enclosed in double quotes. A section name can contain a subsection name in the form *section name:subsection name*.

The **.clink** directive tells the linker to leave the section out of the final object module output of the linker if there are no references found in a linked section to any symbol defined in the specified section. The `--absolute_exe` linker option produces the final output in the form of an absolute, executable output module.

A section in which the entry point of a C program is defined cannot be marked as a conditionally linked section.

Example In this example, the Vars and Counts sections are set for conditional linking.

```

1          *****
2          ** Set Vars section for conditional linking. **
3          *****
4 0000          .sect "Vars"
5          .clink
6 0000 00AA X:  .word 0AAh
7 0002 00AA Y:  .word 0AAh
8 0004 00AA Z:  .word 0AAh
9          *****
10         ** Set Counts section for conditional linking. **
11         *****
12 0000          .sect "Counts"
13         .clink
14 0000 00AA XCount: .word 0AAh
15 0002 00AA YCount: .word 0AAh
16 0004 00AA ZCount: .word 0AAh
17         *****
18         ** .text is unconditionally linked by default. **
19         *****
20 0000          .text
21 0000 403B      MOV  #X_addr, R11
22 0002 0008!
23 0004 4B2B      MOV  @R11, R11
24 0006 5B0C      ADD  R11, R12
25 0008 0000! X_addr: .field X, 32
26 000a 0000
27         *****
28         ** The reference to symbol X causes the Vars **
29         ** section to be linked into the COFF output. **
30         *****

```

.copy/.include
Copy Source File
Syntax

```
.copy ["filename"]
.include ["filename"]
```

Description

The **.copy** and **.include** directives tell the assembler to read source statements from a different file. The statements that are assembled from a copy file are printed in the assembly listing. The statements that are assembled from an included file are *not* printed in the assembly listing, regardless of the number of **.list/.nolist** directives assembled.

When a **.copy** or **.include** directive is assembled, the assembler:

1. Stops assembling statements in the current source file
2. Assembles the statements in the copied/included file
3. Resumes assembling statements in the main source file, starting with the statement that follows the **.copy** or **.include** directive

The *filename* is a required parameter that names a source file. It can be enclosed in double quotes and must follow operating system conventions. If *filename* starts with a number the double quotes are required.

You can specify a full pathname (for example, */320tools/file1.asm*). If you do not specify a full pathname, the assembler searches for the file in:

1. The directory that contains the current source file
2. Any directories named with the **--include_path** assembler option
3. Any directories specified by the **MSP430_A_DIR** environment variable
4. Any directories specified by the **MSP430_C_DIR** environment variable

For more information about the **--include_path** option and **MSP430_A_DIR**, see [Section 3.4](#). For more information about **MSP430_C_DIR**, see the *MSP430 Optimizing C/C++ Compiler User's Guide*.

The **.copy** and **.include** directives can be nested within a file being copied or included. The assembler limits nesting to 32 levels; the host operating system may set additional restrictions. The assembler precedes the line numbers of copied files with a letter code to identify the level of copying. A indicates the first copied file, B indicates a second copied file, etc.

Example 1

In this example, the **.copy** directive is used to read and assemble source statements from other files; then, the assembler resumes assembling into the current file.

The original file, *copy.asm*, contains a **.copy** statement copying the file *byte.asm*. When *copy.asm* assembles, the assembler copies *byte.asm* into its place in the listing (note listing below). The copy file *byte.asm* contains a **.copy** statement for a second file, *word.asm*.

When it encounters the **.copy** statement for *word.asm*, the assembler switches to *word.asm* to continue copying and assembling. Then the assembler returns to its place in *byte.asm* to continue copying and assembling. After completing assembly of *byte.asm*, the assembler returns to *copy.asm* to assemble its remaining statement.

| copy.asm (source file) | byte.asm (first copy file) | word.asm (second copy file) |
|---|---|---|
| <pre>.space 29 .copy "byte.asm" ** Back in original file .string "done"</pre> | <pre>** In byte.asm .byte 32,1+ 'A' .copy "word.asm" ** Back in byte.asm .byte 67h + 3q</pre> | <pre>** In word.asm .word 0ABCDh, 56q</pre> |

Listing file:

```

1 0000          .space 29
2              .copy "byte.asm"
A 1            ** In byte.asm
A 2 001d 0020   .byte 32,1+ 'A'
   001e 0042
A 3            .copy "word.asm"
B 1            ** In word.asm
B 2 0020 ABCD   .word 0ABCDh, 56q
   0022 002E
A 4            ** Back in byte.asm
A 5 0024 006A   .byte 67h + 3q
3
4              ** Back in original file
5 0025 0064    .string "done"
   0026 006F
   0027 006E
   0028 0065
  
```

Example 2

In this example, the `.include` directive is used to read and assemble source statements from other files; then, the assembler resumes assembling into the current file. The mechanism is similar to the `.copy` directive, except that statements are not printed in the listing file.

| include.asm (source file) | byte2.asm (first copy file) | word2.asm (second copy file) |
|---|---|--|
| <pre> .space 29 .include "byte2.asm" ** Back in original file .string "done" </pre> | <pre> ** In byte2.asm .byte 32,1+ 'A' .include "word2.asm" ** Back in byte2.asm .byte 67h + 3q </pre> | <pre> ** In word2.asm .word 0ABCDh, 56q </pre> |

Listing file:

```

1 0000          .space 29
2              .include "byte2.asm"
3
4              **Back in original file
5 0025 0064    .string "done"
   0026 006F
   0027 006E
   0028 0065
  
```

.data
Assemble Into the .data Section
Syntax
.data
Description

The `.data` directive tells the assembler to begin assembling source code into the `.data` section; `.data` becomes the current section. The `.data` section is normally used to contain tables of data or preinitialized variables.

For more information about sections, see [Chapter 2](#).

Example

In this example, code is assembled into the `.data` and `.text` sections.

```

1              ; Comments here
2 0000          .data
3 0000          .space 0xCC
4
5
6              ; Comments here
7 0000          .text
8 0000          INDEX .set 0
9 0000 430B     MOV #INDEX,R11
10
11
12
13             ; Comments here
14 00cc          Table: .data
15 00cc FFFF     .word -1
  
```

```
16 00ce 00FF      .byte 0xFF
17
18
19
20              ; Comments here
21 0002          .text
22 0002 00CC!   con  .field  Table,16
23 0004 421B    MOV   &con,R11
                0006 0002!
24 0008 5B1C    ADD   0(R11),R12
                000a 0000
25
26 00cf          .data
```


.drlist/.drnolist **Control Listing of Directives**
Syntax
.drlist
.drnolist
Description

Two directives enable you to control the printing of assembler directives to the listing file:

The **.drlist** directive enables the printing of all directives to the listing file.

The **.drnolist** directive suppresses the printing of the following directives to the listing file. The **.drnolist** directive has no affect within macros.

- .asg
- .break
- .emsg
- .eval
- .fclist
- .fcnolist
- .mlist
- .mmsg
- .mnolist
- .sslist
- .ssnolist
- .var
- .wmsg

By default, the assembler acts as if the **.drlist** directive had been specified.

Example

This example shows how **.drnolist** inhibits the listing of the specified directives.

Source file:

```
.asg    0, x
.loop   2
.eval   x+1, x
.endloop
.drnolist
.asg    1, x
.loop   3
.eval   x+1, x
.endloop
```

Listing file:

```
1          .asg    0, x
2          .loop   2
3          .eval   x+1, x
4          .endloop
1          .eval   0+1, x
1          .eval   1+1, x
5
6          .drnolist
7
9          .loop   3
10         .eval   x+1, x
11        .endloop
```

Listing file:

.emsg/.mmsg/.wmsg *Define Messages*

Syntax

```
.emsg string
.mmsg string
.wmsg string
```

Description These directives allow you to define your own error and warning messages. When you use these directives, the assembler tracks the number of errors and warnings it encounters and prints these numbers on the last line of the listing file.

The **.emsg** directive sends an error message to the standard output device in the same manner as the assembler. It increments the error count and prevents the assembler from producing an object file.

The **.mmsg** directive sends an assembly-time message to the standard output device in the same manner as the **.emsg** and **.wmsg** directives. It does not, however, set the error or warning counts, and it does not prevent the assembler from producing an object file.

The **.wmsg** directive sends a warning message to the standard output device in the same manner as the **.emsg** directive. It increments the warning count rather than the error count, however. It does not prevent the assembler from producing an object file.

Example In this example, the message ERROR -- MISSING PARAMETER is sent to the standard output device.

Source file:

```
MSG_EX .macro parm1
      .if    $$symlen(parm1) = 0
      .emsg "ERROR -- MISSING PARAMETER"
      .else
      ADD   parm1, r7, r8
      .endif
      .endm
MSG_EX R11
MSG_EX
```

Listing file:

```

1          MSG_EX .macro parm1
2          .if    $symlen(parm1)=0
3          .emsg "ERROR -- MISSING PARAMETER"
4          .else
5          ADD   parm1, r7
6          .endif
7          .endm
8
9 0000          MSG_EX R11
1         .if    $symlen(parm1)=0
1         .emsg "ERROR -- MISSING PARAMETER"
1         .else
1         ADD   R11, r7
1         .endif
10
11 0002          MSG_EX
1         .if    $symlen(parm1)=0
1         .emsg "ERROR -- MISSING PARAMETER"
"emsg.asm", ERROR!   at line 11: [ ***** USER ERROR ***** - ] ERROR --
MISSING PARAMETER
1         .else
1         ADD   parm1, r7
1         .endif
1 Assembly Error, No Assembly Warnings
```

In addition, the following messages are sent to standard output by the assembler:

```
*** ERROR!   line 11: ***** USER ERROR ***** - : ERROR -- MISSING PARAMETER
             .emsg "ERROR -- MISSING PARAMETER"
```

```

1 Error, No Warnings
Errors in source - Assembler Aborted

```

.end

End Assembly

Syntax

.end

Description

The **.end** directive is optional and terminates assembly. The assembler ignores any source statements that follow a **.end** directive. If you use the **.end** directive, it must be the last source statement of a program.

This directive has the same effect as an end-of-file character. You can use **.end** when you are debugging and you want to stop assembling at a specific point in your code.

Ending a Macro

Note: Do not use the **.end** directive to terminate a macro; use the **.endm** macro directive instead.

Example

This example shows how the **.end** directive terminates assembly. If any source statements follow the **.end** directive, the assembler ignores them.

Source file:

```

START:  .space 300
TEMP    .set  15
        .bss  LOC1,0x48
LOCL_n  .word  LOC1
        MOV   #TEMP,R11
        MOV   &LOCL_n,R12
        MOV   0(R12),R13
        .end
        .byte 4
        .word 0xCCC

```

Listing file:

```

1 0000          START:  .space 300
2      000F  TEMP    .set  15
3 0000          .bss  LOC1,0x48
4 012c 0000!  LOCL_n  .word  LOC1
5 012e 403B          MOV   #TEMP,R11
   0130 000F
6 0132 421C          MOV   &LOCL_n,R12
   0134 012C!
7 0136 4C1D          MOV   0(R12),R13
   0138 0000
8                          .end

```

.fclist/.fcno list **Control Listing of False Conditional Blocks**

Syntax
.fclist
.fcno list
Description

Two directives enable you to control the listing of false conditional blocks:

The **.fclist** directive allows the listing of false conditional blocks (conditional blocks that do not produce code).

The **.fcno list** directive suppresses the listing of false conditional blocks until a **.fclist** directive is encountered. With **.fcno list**, only code in conditional blocks that are actually assembled appears in the listing. The **.if**, **.elseif**, **.else**, and **.endif** directives do not appear.

By default, all conditional blocks are listed; the assembler acts as if the **.fclist** directive had been used.

Example

This example shows the assembly language and listing files for code with and without the conditional blocks listed.

Source file:

```

AAA .set 1
BB .set 0
.fclist
.if AAA
ADD #1024,R11
.else
ADD #1024*10,R11
.endif
.fcno list
.if AAA
ADD #1024,R11
.else
ADD #1024*10,R11
.endif

```

Listing file:

```

1      0001  AAA      .set 1
2      0000  BB       .set 0
3                               .fclist
4                               .if  AAA
5 0000 503B      ADD   #1024,R11
      0002 0400
6                               .else
7                               ADD   #1024*10,R11
8                               .endif
9                               .fcno list
11 0004 503B      ADD   #1024,R11
      0006 0400

```

.field
Initialize Field
Syntax
.field *value* [, *size in bits*]

Description

The **.field** directive initializes a multiple-bit field within a single word (16 bits) of memory. This directive has two operands:

- The *value* is a required parameter; it is an expression that is evaluated and placed in the field. The value must be absolute.
- The *size in bits* is an optional parameter; it specifies a number from 1 to 32, which is the number of bits in the field. If you do not specify a size, the assembler assumes the size is 16 bits. If you specify a value that cannot fit in *size in bits*, the assembler truncates the value and issues a warning message. For example, **.field 3,1** causes the assembler to truncate the value 3 to 1; the assembler also prints the message:

```
*** WARNING! line 21: W0001: Field value truncated to 1
      .field 3, 1
```

Successive **.field** directives pack values into the specified number of bits.

You can use the **.align** directive to force the next **.field** directive to begin packing into a new word.

If you use a label, it points to the byte that contains the specified field.

When you use **.field** in a **.struct/.endstruct** sequence, **.field** defines a member's size; it does not initialize memory. For more information, see the [.struct/.endstruct/.tag](#) topic .

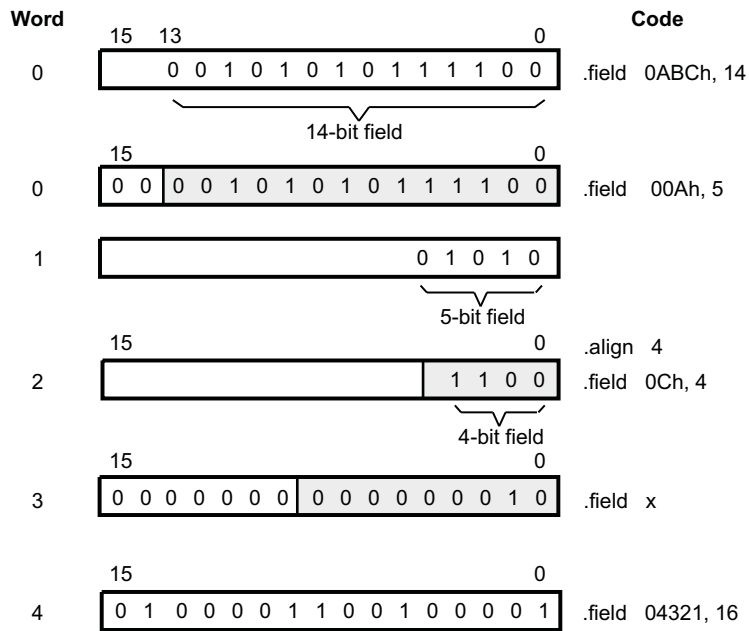
Example

This example shows how fields are packed into a word. The SPC does not change until a word is filled and the next word is begun. [Figure 4-6](#) shows how the directives in this example affect memory.

```

1          *****
2          **      Initialize a 14-bit field.  **
3          *****
4 0000 0ABC      .field  0ABCh, 14
5
6          *****
7          **      Initialize a 5-bit field    **
8          **      in the same word.         **
9          *****
10 0002 000A L_F: .field  0Ah, 5
11
12          *****
13          **      Write out the word.       **
14          *****
15          .align 4
16
17          *****
18          **      Initialize a 4-bit field.  **
19          **      This fields starts a new word. **
20          *****
21 0004 000C x:   .field  0Ch, 4
22
23          *****
24          **      16-bit relocatable field  **
25          **      in the next word.         **
26          *****
27 0006 0004!   .field  x
28
29          *****
30          **      Initialize a 16-bit field.  **
31          *****
32 0008 4321   .field  04321h, 16
```


Figure 4-6. The .field Directive



.global/.def/.ref
Identify Global Symbols
Syntax

```
.global symbol1[, ..., symboln]
```

```
.def symbol1[, ..., symboln]
```

```
.ref symbol1[, ... , symboln]
```

Description

Three directives identify global symbols that are defined externally or can be referenced externally:

The **.def** directive identifies a symbol that is defined in the current module and can be accessed by other files. The assembler places this symbol in the symbol table.

The **.ref** directive identifies a symbol that is used in the current module but is defined in another module. The linker resolves this symbol's definition at link time.

The **.global** directive acts as a **.ref** or a **.def**, as needed.

A global *symbol* is defined in the same manner as any other symbol; that is, it appears as a label or is defined by the **.set**, **.equ**, **.bss**, or **.usect** directive. As with all symbols, if a global symbol is defined more than once, the linker issues a multiple-definition error. The **.ref** directive always creates a symbol table entry for a symbol, whether the module uses the symbol or not; **.global**, however, creates an entry only if the module actually uses the symbol.

A symbol can be declared global for either of two reasons:

- If the symbol is *not defined in the current module* (which includes macro, copy, and include files), the **.global** or **.ref** directive tells the assembler that the symbol is defined in an external module. This prevents the assembler from issuing an unresolved reference error. At link time, the linker looks for the symbol's definition in other modules.
- If the symbol is *defined in the current module*, the **.global** or **.def** directive declares that the symbol and its definition can be used externally by other modules. These types of references are resolved at link time.

Example

This example shows four files. The **file1.lst** and **file2.lst** refer to each other for all symbols used; **file3.lst** and **file4.lst** are similarly related.

The **file1.lst** and **file3.lst** files are equivalent. Both files define the symbol **INIT** and make it available to other modules; both files use the external symbols **X**, **Y**, and **Z**. Also, **file1.lst** uses the **.global** directive to identify these global symbols; **file3.lst** uses **.ref** and **.def** to identify the symbols.

The **file2.lst** and **file4.lst** files are equivalent. Both files define the symbols **X**, **Y**, and **Z** and make them available to other modules; both files use the external symbol **INIT**. Also, **file2.lst** uses the **.global** directive to identify these global symbols; **file4.lst** uses **.ref** and **.def** to identify the symbols.

file1.lst

```

1           ; Global symbol defined in this file
2           .global INIT
3           ; Global symbols defined in file2.lst
4           .global X, Y, Z
5 0000      INIT:
6 0000 503B      ADD      #56h, R11
7 0002 0056
7 0004 0000!      .word   X
8           ;
9           ;
10          ;
11          .end

```

file2.lst

```

1          ; Global symbols defined in this file
2          .global X, Y, Z
3          ; Global symbol defined in file1.lst
4          .global INIT
5 0001     X:      .set    1
6 0002     Y:      .set    2
7 0003     Z:      .set    3
8 0000 0000! .word  INIT
9          ;      .
10         ;      .
11         ;      .
12         .end

```

file3.lst

```

1          ; Global symbol defined in this file
2          .def  INIT
3          ; Global symbols defined in file2.lst
4          .ref  X, Y, Z
5 0000     INIT:
6 0000 503B      ADD    #56h, R11
7 0002 0056
8 0004 0000!     .word  X
9          ;      .
10         ;      .
11         .end

```

file4.lst

```

1          ; Global symbols defined in this file
2          .def  X, Y, Z
3          ; Global symbol defined in file3.lst
4          .ref  INIT
5 0001     X:      .set    1
6 0002     Y:      .set    2
7 0003     Z:      .set    3
8 0000 0000!     .word  INIT
9          ;      .
10         ;      .
11         ;      .
12         .end

```

.half/.short
Initialize 16-Bit Integers
Syntax

```
.half value1[, ... , valuen]
```

```
.short value1[, ... , valuen]
```

Description

The **.half** and **.short** directives place one or more values into consecutive halfwords in the current section. A *value* can be either:

- An expression that the assembler evaluates and treats as a 16-bit signed or unsigned number
- A character string enclosed in double quotes. Each character in a string represents a separate value and is stored alone in the least significant eight bits of a 16-bit field, which is padded with 0s.

The assembler truncates values greater than 16 bits. You can use as many values as fit on a single line, but the total line length cannot exceed 200 characters.

If you use a label with **.half** or **.short**, it points to the location where the assembler places the first byte.

These directives perform a word (16-bit) alignment before data is written to the section.

When you use **.half** or **.short** in a **.struct/.endstruct** sequence, they define a member's size; they do not initialize memory. For more information, see the [.struct/.endstruct/.tag topic](#).

Example

In this example, **.half** is used to place 16-bit values (10, -1, abc, and a) into consecutive halfwords in memory; **.short** is used to place 16-bit values (8, -3, def, and b) into consecutive halfwords in memory. The label STRN has the value 100ch, which is the location of the first initialized halfword for **.short**.

```

1 0000                .space 100h * 16
2 1000 000A          .half 10, -1, "abc", "a"
   1002 FFFF
   1004 0061
   1006 0062
   1008 0063
   100a 0061
3 100c 0008 STRN     .short 8, -3, "def", 'b'
   100e FFFD
   1010 0064
   1012 0065
   1014 0066
   1016 0062

```

.if/.elseif/.else/.endif Assemble Conditional Blocks

Syntax

```
.if well-defined expression
[.elseif well-defined expression]
[.else]
.endif
```

Description

Four directives provide conditional assembly:

The **.if** directive marks the beginning of a conditional block. The *well-defined expression* is a required parameter.

- If the expression evaluates to true (nonzero), the assembler assembles the code that follows the expression (up to a **.elseif**, **.else**, or **.endif**).
- If the expression evaluates to false (0), the assembler assembles code that follows a **.elseif** (if present), **.else** (if present), or **.endif** (if no **.elseif** or **.else** is present).

The **.elseif** directive identifies a block of code to be assembled when the **.if** expression is false (0) and the **.elseif** expression is true (nonzero). When the **.elseif** expression is false, the assembler continues to the next **.elseif** (if present), **.else** (if present), or **.endif** (if no **.elseif** or **.else** is present). The **.elseif** directive is optional in the conditional block, and more than one **.elseif** can be used. If an expression is false and there is no **.elseif** statement, the assembler continues with the code that follows a **.else** (if present) or a **.endif**.

The **.else** directive identifies a block of code that the assembler assembles when the **.if** expression and all **.elseif** expressions are false (0). The **.else** directive is optional in the conditional block; if an expression is false and there is no **.else** statement, the assembler continues with the code that follows the **.endif**.

The **.endif** directive terminates a conditional block.

The **.elseif** and **.else** directives can be used in the same conditional assembly block, and the **.elseif** directive can be used more than once within a conditional assembly block.

See [Section 3.9.4](#) for information about relational operators.

Example

This example shows conditional assembly:

```

1      0001  SYM1  .set    1
2      0002  SYM2  .set    2
3      0003  SYM3  .set    3
4      0004  SYM4  .set    4
5
6          If_4:  .if     SYM4 = SYM2 * SYM2
7 0000 0004      .byte   SYM4          ; Equal values
8          .else
9          .byte   SYM2 * SYM2      ; Unequal values
10         .endif
11
12          If_5:  .if     SYM1 <= 10
13 0001 000A      .byte   10          ; Equal values
14         .else
15         .byte   SYM1            ; Unequal values
16         .endif
17
18          If_6:  .if     SYM3 * SYM2 != SYM4 + SYM2
19         .byte   SYM3 * SYM2      ; Unequal value
20         .else
21 0002 0008      .byte   SYM4 + SYM4      ; Equal values
22         .endif
23
24          If_7:  .if     SYM1 = SYM2
25         .byte   SYM1
26         .elseif  SYM2 + SYM3 = 5
27 0003 0005      .byte   SYM2 + SYM3
28         .endif
```

.int/.long
Initialize 16-BitIntegers
Syntax

```
.int value1[, ... , valuen]
```

```
.long value1[, ... , valuen]
```

Description

The **.int** and **.long** directives place one or more values into consecutive words in the current section. Each value is placed in a 16-bit word by itself and is aligned on a word boundary. A *value* can be either:

- An expression that the assembler evaluates and treats as a 16-bit signed or unsigned number
- A character string enclosed in double quotes. Each character in a string represents a separate value and is stored alone in the least significant eight bits of a 16-bit field, which is padded with 0s.

A value can be either an absolute or a relocatable expression. If an expression is relocatable, the assembler generates a relocation entry that refers to the appropriate symbol; the linker can then correctly patch (relocate) the reference. This allows you to initialize memory with pointers to variables or labels.

You can use as many values as fit on a single line (200 characters). If you use a label with these directives, it points to the first word that is initialized.

When you use these directives in a **.struct/.endstruct** sequence, they define a member's size; they do not initialize memory. See the [.struct/.endstruct/tag](#) topic .

Example

In this example, the **.int** and **.word** directives are used to initialize words. The symbol **WORDX** points to the first word that is reserved by **.word**.

```

1 0000                .space 73h
2 0000                .bss PAGE, 128
3 0080                .bss SYMPTR, 4
4 0074 403B INST: MOV  #056h, R11
   0076 0056
5 0078 000A          .int  10, SYMPTR, -1, 35 + 'a', INST, "abc"
   007a 0080!
   007c FFFF
   007e 0084
   0080 0074!
   0082 0061
   0084 0062
   0086 0063
6 0000 0C80 WORDX: .word 3200, 1 + 'AB', -0AFh, 'X'
   0002 4242
   0004 FF51
   0006 0058

```

.label *Create a Load-Time Address Label*

Syntax `.label symbol`

Description The `.label` directive defines a special *symbol* that refers to the load-time address rather than the run-time address within the current section. Most sections created by the assembler have relocatable addresses. The assembler assembles each section as if it started at 0, and the linker relocates it to the address at which it loads and runs.

For some applications, it is desirable to have a section load at one address and run at a *different* address. For example, you may want to load a block of performance-critical code into slower memory to save space and then move the code to high-speed memory to run it. Such a section is assigned two addresses at link time: a load address and a run address. All labels defined in the section are relocated to refer to the run-time address so that references to the section (such as branches) are correct when the code runs.

The `.label` directive creates a special label that refers to the *load-time* address. This function is useful primarily to designate where the section was loaded for purposes of the code that relocates the section.

Example This example shows the use of a load-time address label.

```
sect ".examp"
    .label examp_load ; load address of section
start:
    ; run address of section
    <code>
finish:
    ; run address of section end
    .label examp_end ; load address of section end
```

See [Section 7.9](#) for more information about assigning run-time and load-time addresses in the linker.

.length/.width
Set Listing Page Size
Syntax

.length [*page length*]

.width [*page width*]

Description

Two directives allow you to control the size of the output listing file.

The **.length** directive sets the page length of the output listing file. It affects the current and following pages. You can reset the page length with another **.length** directive.

- Default length: 60 lines. If you do not use the **.length** directive or if you use the **.length** directive without specifying the *page length*, the output listing length defaults to 60 lines.
- Minimum length: 1 line
- Maximum length: 32 767 lines

The **.width** directive sets the page width of the output listing file. It affects the next line assembled and the lines following. You can reset the page width with another **.width** directive.

- Default width: 132 characters. If you do not use the **.width** directive or if you use the **.width** directive without specifying a *page width*, the output listing width defaults to 132 characters.
- Minimum width: 80 characters
- Maximum width: 200 characters

The width refers to a full line in a listing file; the line counter value, SPC value, and object code are counted as part of the width of a line. Comments and other portions of a source statement that extend beyond the page width are truncated in the listing.

The assembler does not list the **.width** and **.length** directives.

Example

The following example shows how to change the page length and width.

```
*****
**          Page length = 65 lines          **
**          Page width = 85 characters      **
*****
          .length    65
          .width     85

*****
**          Page length = 55 lines          **
**          Page width = 100 characters     **
*****
          .length    55
          .width     100
```


.list/.nolist
Start/Stop Source Listing

Syntax

.list
.nolist

Description

Two directives enable you to control the printing of the source listing:

The **.list** directive allows the printing of the source listing.

The **.nolist** directive suppresses the source listing output until a **.list** directive is encountered. The **.nolist** directive can be used to reduce assembly time and the source listing size. It can be used in macro definitions to suppress the listing of the macro expansion.

The assembler does not print the **.list** or **.nolist** directives or the source statements that appear after a **.nolist** directive. However, it continues to increment the line counter. You can nest the **.list/.nolist** directives; each **.nolist** needs a matching **.list** to restore the listing.

By default, the source listing is printed to the listing file; the assembler acts as if the **.list** directive had been used. However, if you do not request a listing file when you invoke the assembler by including the **--asm_listing** option on the command line (see [Section 3.3](#)), the assembler ignores the **.list** directive.

Example

This example shows how the **.copy** directive inserts source statements from another file. The first time this directive is encountered, the assembler lists the copied source lines in the listing file. The second time this directive is encountered, the assembler does not list the copied source lines, because a **.nolist** directive was assembled. The **.nolist**, the second **.copy**, and the **.list** directives do not appear in the listing file. Also, the line counter is incremented, even when source statements are not listed.

Source file:

```
.copy "copy2.asm"
* Back in original file
NOP
.nolist
.copy "copy2.asm"
.list
* Back in original file
.string "Done"
```

Listing file:

```
1 .copy "copy2.asm"
A 1 * In copy2.asm (copy file)
A 2 0000 0020 .word 32, 1 + 'A'
0002 0042
2 * Back in original file
3 0004 4303 NOP
7 * Back in original file
8 000a 0044 .string "Done"
000b 006F
000c 006E
000d 0065
```

.long
Initialize 32-Bit Integer
Syntax

```
.long value1[, ... , valuen]
```

Description

The `.long` directive places one or more values into consecutive words in the current section. A value can be either:

- An expression that the assembler evaluates and treats as a 32-bit signed or unsigned number
- A character string enclosed in double quotes. Each character in a string represents a separate value and is stored alone in the least significant eight bits of a 32-bit field, which is padded with 0s.

A value can be either an absolute or a relocatable expression. If an expression is relocatable, the assembler generates a relocation entry that refers to the appropriate symbol; the linker can then correctly patch (relocate) the reference. This allows you to initialize memory with pointers to variables or labels.

The `.long` directive performs a word (16-bit) alignment before any data is written to the section.

When you use `.long` directive in a `.struct/.endstruct` sequence, it defines a member's size; it does not initialize memory. See the [.struct/.endstruct/.tag topic](#).

Example

This example shows how the `.long` directive initializes words. The symbol `DAT1` points to the first word that is reserved.

```

1 0000 ABCD  DAT1:  .long  0ABCDh, 'A' + 100h, 'g', 'o'
   0002 0000
   0004 0141
   0006 0000
   0008 0067
   000a 0000
   000c 006F
   000e 0000
2 0010 0000!      .long  DAT1, 0AABBCCDDh
   0012 0000
   0014 CCDD
   0016 AABB
3 0018          DAT2:
```

.loop/.endloop/.break Assemble Code Block Repeatedly

Syntax

```

.loop [well-defined expression]
.break [well-defined expression]
.endloop
  
```

Description Three directives allow you to repeatedly assemble a block of code:

The **.loop** directive begins a repeatable block of code. The optional expression evaluates to the loop count (the number of loops to be performed). If there is no *well-defined expression*, the loop count defaults to 1024, unless the assembler first encounters a **.break** directive with an expression that is true (nonzero) or omitted.

The **.break** directive, along with its expression, is optional. This means that when you use the **.loop** construct, you do not have to use the **.break** construct. The **.break** directive terminates a repeatable block of code only if the *well-defined expression* is true (nonzero) or omitted, and the assembler breaks the loop and assembles the code after the **.endloop** directive. If the expression is false (evaluates to 0), the loop continues.

The **.endloop** directive terminates a repeatable block of code; it executes when the **.break** directive is true (nonzero) or when the number of loops performed equals the loop count given by **.loop**.

Example This example illustrates how these directives can be used with the **.eval** directive. The code in the first six lines expands to the code immediately following those six lines.

```

      1          .eval    0,x
      2          COEF .loop
      3          .word    x*100
      4          .eval    x+1, x
      5          .break   x = 6
      6          .endloop
1 0000 0000      .word    0*100
1          .eval    0+1, x
1          .break   1 = 6
1 0002 0064      .word    1*100
1          .eval    1+1, x
1          .break   2 = 6
1 0004 00C8      .word    2*100
1          .eval    2+1, x
1          .break   3 = 6
1 0006 012C      .word    3*100
1          .eval    3+1, x
1          .break   4 = 6
1 0008 0190      .word    4*100
1          .eval    4+1, x
1          .break   5 = 6
1 000a 01F4      .word    5*100
1          .eval    5+1, x
1          .break   6 = 6
  
```

.macro/.endm
Define Macro

Syntax

```

macname .macro [parameter1 [, ... parametern]]
           model statements or macro directives
           .endm

```

Description

The **.macro** and **.endm** directives are used to define macros.

You can define a macro anywhere in your program, but you must define the macro before you can use it. Macros can be defined at the beginning of a source file, in an `.include/.copy` file, or in a macro library.

| | |
|-------------------------|--|
| <i>macname</i> | names the macro. You must place the name in the source statement's label field. |
| .macro | identifies the source statement as the first line of a macro definition. You must place .macro in the opcode field. |
| [<i>parameters</i>] | are optional substitution symbols that appear as operands for the .macro directive. |
| <i>model statements</i> | are instructions or assembler directives that are executed each time the macro is called. |
| <i>macro directives</i> | are used to control macro expansion. |
| .endm | marks the end of the macro definition. |

Macros are explained in further detail in [Chapter 5](#).

.mlib
Define Macro Library
Syntax
.mlib ["filename"]

Description

The **.mlib** directive provides the assembler with the *filename* of a macro library. A macro library is a collection of files that contain macro definitions. The macro definition files are bound into a single file (called a library or archive) by the archiver.

Each file in a macro library contains one macro definition that corresponds to the name of the file. The *filename* of a macro library member must be the same as the macro name, and its extension must be **.asm**. The filename must follow host operating system conventions; it can be enclosed in double quotes. You can specify a full pathname (for example, `c:\320tools\macs.lib`). If you do not specify a full pathname, the assembler searches for the file in the following locations in the order given:

1. The directory that contains the current source file
2. Any directories named with the `--include_path` assembler option
3. Any directories specified by the `MSP430_A_DIR` environment variable
4. Any directories specified by the `MSP430_C_DIR` environment variable

See [Section 3.4](#) for more information about the `--include_path` option.

When the assembler encounters a **.mlib** directive, it opens the library specified by the *filename* and creates a table of the library's contents. The assembler enters the names of the individual library members into the opcode table as library entries. This redefines any existing opcodes or macros that have the same name. If one of these macros is called, the assembler extracts the entry from the library and loads it into the macro table. The assembler expands the library entry in the same way it expands other macros, but it does not place the source code into the listing. Only macros that are actually called from the library are extracted, and they are extracted only once.

See [Chapter 5](#) for more information on macros and macro libraries.

Example

The code creates a macro library that defines two macros, `inc4.asm` and `dec4.asm`. The file `inc4.asm` contains the definition of `inc4` and `dec4.asm` contains the definition of `dec4`.

| inc4.asm | dec4.asm |
|---|---|
| <pre>* Macro for incrementing inc4 .macro reg ADD.W #1, reg ADD.W #1, reg ADD.W #1, reg ADD.W #1, reg .endm</pre> | <pre>* Macro for decrementing dec4 .macro reg SUB.W #1, reg SUB.W #1, reg SUB.W #1, reg SUB.W #1, reg .endm</pre> |

Use the archiver to create a macro library:

```
ar430 -a mac inc4.asm dec4.asm
```

Now you can use the **.mlib** directive to reference the macro library and define the `inc4.asm` and `dec4.asm` macros:

```

1          .mlist
2          .mlib "mac.lib"
3          ; Macro call
4 0000      inc4 R11, R12, R13, R14
1          0000 531B      ADD.W #1,R11
1          0002 531C      ADD.W #1,R12
1          0004 531D      ADD.W #1,R13
1          0006 531E      ADD.W #1,R14
5
6          ; Macro call
7 0008      dec4 R11, R12, R13, R14
1          0008 831B      SUB.W #1,R11
1          000a 831C      SUB.W #1,R12
1          000c 831D      SUB.W #1,R13
1          000e 831E      SUB.W #1,R14
```

.mlist/.mno1ist **Start/Stop Macro Expansion Listing**

Syntax
.m1ist
.mno1ist
Description

Two directives enable you to control the listing of macro and repeatable block expansions in the listing file:

The **.m1ist** directive allows macro and **.loop/.endloop** block expansions in the listing file.

The **.mno1ist** directive suppresses macro and **.loop/.endloop** block expansions in the listing file.

By default, the assembler behaves as if the **.m1ist** directive had been specified.

See [Chapter 5](#) for more information on macros and macro libraries. See the [.loop/.break/.endloop topic](#) for information on conditional blocks.

Example

This example defines a macro named STR_3. The first time the macro is called, the macro expansion is listed (by default). The second time the macro is called, the macro expansion is not listed, because a **.mno1ist** directive was assembled. The third time the macro is called, the macro expansion is again listed because a **.m1ist** directive was assembled.

```

1          STR_3  .macro   P1, P2, P3
2              .string  ":p1:", ":p2:", ":p3:"
3              .endm
4
5 0000          STR_3 "as", "I", "am"   ; Invoke STR_3 macro.
1 0000 003A      .string  ":p1:", ":p2:", ":p3:"
0001 0070
0002 0031
0003 003A
0004 003A
0005 0070
0006 0032
0007 003A
0008 003A
0009 0070
000a 0033
000b 003A
6          .mno1ist          ; Suppress expansion.
7 000c      STR_3 "as", "I", "am"   ; Invoke STR_3 macro.
8          .m1ist           ; Show macro expansion.
9 0018      STR_3 "as", "I", "am"   ; Invoke STR_3 macro.
1 0018 003A      .string  ":p1:", ":p2:", ":p3:"
0019 0070
001a 0031
001b 003A
001c 003A
001d 0070
001e 0032
001f 003A
0020 003A
0021 0070
0022 0033
0023 003A

```

.newblock
Terminate Local Symbol Block

Syntax
.newblock
Description

The **.newblock** directive undefines any local labels currently defined. Local labels, by nature, are temporary; the **.newblock** directive resets them and terminates their scope.

A local label is a label in the form $\$n$, where n is a single decimal digit, or $name?$, where $name$ is a legal symbol name. Unlike other labels, local labels are intended to be used locally, cannot be used in expressions, and do not qualify for branch expansion if used with a branch. They can be used only as operands in 8-bit jump instructions. Local labels are not included in the symbol table.

After a local label has been defined and (perhaps) used, you should use the **.newblock** directive to reset it. The **.text**, **.data**, and **.sect** directives also reset local labels. Local labels that are defined within an include file are not valid outside of the include file.

See [Section 3.8.2](#) for more information on the use of local labels.

Example

This example shows how the local label $\$1$ is declared, reset, and then declared again.

```

1          .global ADDRA,ADDRB,ADDRC
2
3 0000 403B Label1: MOV #ADDRA, R11 ; Load Address A to R11
   0002 0000!
4 0004 803B          SUB #ADDRB, R11 ; Subtract Address B.
   0006 0000!
5 0008 3803          JL $1          ; If < 0, branch to $1
6 000a 403B          MOV #ADDRB, R11 ; otherwise, load ADDRB to R11
   000c 0000!
7 000e 3C02          JMP $2          ; and branch to $2
8 0010 403B $1       MOV #ADDRA, R11 ; $1: load ADDRA to AC0.
   0012 0000!
9 0014 503B $2       ADD #ADDRC, R11 ; $2: add ADDRc.
   0016 0000!
10         .newblock        ; Undefine $1 so can be used again.
11 0018 3C02          JMP $1          ; If less than zero,branch to $1.
12 001a 4B82          MOV R11,&ADDRc ; Store AC0 low in ADDRc.
   001c 0000!
13 001e 4303 $1       NOP

```

.option
Select Listing Options
Syntax
.option *option*₁[, *option*₂, . . .]

Description

The **.option** directive selects options for the assembler output listing. The *options* must be separated by commas; each option selects a listing feature. These are valid options:

- A** turns on listing of all directives and data, and subsequent expansions, macros, and blocks.
- B** limits the listing of **.byte** and **.char** directives to one line.
- H** limits the listing of **.half** and **.short** directives to one line.
- L** limits the listing of **.long** directives to one line.
- M** turns off macro expansions in the listing.
- N** turns off listing (performs **.nolist**).
- O** turns on listing (performs **.list**).
- R** resets the **B**, **H**, **M**, **T**, and **W** (turns off the limits of **B**, **H**, **M**, **T**, and **W**).
- T** limits the listing of **.string** directives to one line.
- W** limits the listing of **.word** and **.int** directives to one line.
- X** produces a cross-reference listing of symbols. You can also obtain a cross-reference listing by invoking the assembler with the **--cross_reference** option (see [Section 3.3](#)).

Options *are not* case sensitive.

Example

This example shows how to limit the listings of the **.byte**, **.char**, **.int**, **.long**, **.word**, and **.string** directives to one line each.

```

1          *****
2          ** Limit the listing of .byte, .char, .int, .long, **
3          ** .word, and .string directives to 1 line each. **
4          *****
5          .option B, W, T
6 0000 00BD .byte  -'C', 0B0h, 5
7 0003 00BC .char  -'D', 0C0h, 6
8 0006 000A .int   10, 35 + 'a', "abc"
9 0010 CCDD .long  0AABBCCDDh, 536 + 'A'
10         0012 AABB
11         0014 0259
12         0016 0000
13         0018 15AA .word  5546, 78h
14         001c 0045 .string "Extended Registers"
15
16         *****
17         ** Reset the listing options. **
18         *****
19         .option R
20 002e 00BD .byte  -'C', 0B0h, 5
21         002f 00B0
22         0030 0005
23         0031 00BC .char  -'D', 0C0h, 6
24         0032 00C0
25         0033 0006
26         0034 000A .int   10, 35 + 'a', "abc"
27         0036 0084
28         0038 0061
29         003a 0062
30         003c 0063
31         003e CCDD .long  0AABBCCDDh, 536 + 'A'
32         0040 AABB
33         0042 0259
34         0044 0000
35         0046 15AA .word  5546, 78h

```



```

    0048 0078
22 004a 0045      .string "Extended Registers"
    004b 0078
    004c 0074
    004d 0065
    004e 006E
    004f 0064
    0050 0065
    0051 0064
    0052 0020
    0053 0052
    0054 0065
    0055 0067
    0056 0069
    0057 0073
    0058 0074
    0059 0065
    005a 0072
    005b 0073
  
```

.page *Eject Page in Listing*

Syntax `.page`

Description The **.page** directive produces a page eject in the listing file. The **.page** directive is not printed in the source listing, but the assembler increments the line counter when it encounters the **.page** directive. Using the **.page** directive to divide the source listing into logical divisions improves program readability.

Example This example shows how the **.page** directive causes the assembler to begin a new page of the source listing.

Source file:

```

    .title      "**** Page Directive Example ****"
;             .
;             .
;             .
    .page
  
```

Listing file:

```

MSP430 COFF Assembler PC vx.x.x      Day      Time      Year

Tools Copyright (c) 2003-2009 Texas Instruments Incorporated
**** Page Directive Example ****                                           PAGE 1

    2             ;             .
    3             ;             .
    4             ;             .
MSP430 COFF Assembler PC vx.x.x      Day      Time      Year

Tools Copyright (c) 2003-2009 Texas Instruments Incorporated
**** Page Directive Example ****                                           PAGE 2
  
```

.sect *Assemble Into Named Section*
Syntax `.sect "section name"`
Description The `.sect` directive defines a named section that can be used like the default `.text` and `.data` sections. The `.sect` directive tells the assembler to begin assembling source code into the named section.

The *section name* identifies the section. The section name is significant to 200 characters and must be enclosed in double quotes. A section name can contain a subsection name in the form *section name:subsection name*.

See [Chapter 2](#) for more information about sections.

Example This example defines a special-purpose section named `Vars` and assembles code into it.

```

1          *****
2          **      Begin assembling into .text section.      **
3          *****
4 0000          .text
5 0000 403B      MOV #0x78,R11
   0002 0078
6 0004 503B      ADD #0x78,R11
   0006 0078
7          *****
8          **      Begin assembling into Vars section.      **
9          *****
10 0000          .sect Vars
11 0000 CCCD      .float 0.05
   0002 3D4C
12 0004 00AA      X: .word 0xAA
13          *****
14          **      Resume assembling into .text section.      **
15          *****
16 0008          .text
17 0008 5B0C      ADD R11,R12
18          *****
19          **      Resume assembling into Vars section.      **
20          *****
21 0006          .sect Vars
22 0006 000D      .field 13
23 0008 000A      .field 0xA
24 000a 0010      .field 0x10

```

.set/.equ *Define Assembly-Time Constant*

Syntax *symbol .set value*

symbol .equ value

Description

The **.set** and **.equ** directives equate a constant value to a symbol. The symbol can then be used in place of a value in assembly source. This allows you to equate meaningful names with constants and other values. The **.set** and **.equ** directives are identical and can be used interchangeably.

- The *symbol* is a label that must appear in the label field.
- The *value* must be a well-defined expression, that is, all symbols in the expression must be previously defined in the current source module.

Undefined external symbols and symbols that are defined later in the module cannot be used in the expression. If the expression is relocatable, the symbol to which it is assigned is also relocatable.

The value of the expression appears in the object field of the listing. This value is not part of the actual object code and is not written to the output file.

Symbols defined with **.set** or **.equ** can be made externally visible with the **.def** or **.global** directive (see the [.global/.def/.ref topic](#)). In this way, you can define global absolute constants.

Example

This example shows how symbols can be assigned with **.set** and **.equ**.

```

1          *****
2          **   Equate symbol ACCUM to register R11 and use   **
3          **               it instead of the register.       **
4          *****
5 000B     ACCUM   .set R11
6 0000 401B     MOV 0x56, ACCUM
   0002 0054
7
8          *****
9          **   Set symbol INDEX to an integer expression   **
10         **               and use it as an immediate operand. **
11         *****
12 0035     INDEX   .equ 100/2 + 3
13 0004 503B     ADD #INDEX, ACCUM
   0006 0035
14
15         *****
16         **   Set symbol SYMTAB to a relocatable expression **
17         **               and use it as a relocatable operand. **
18         *****
19 0008 000A     LABEL .word 10
20           0009! SYMTAB .set LABEL + 1
21
22         *****
23         **   Set symbol NSYMS equal to the symbol INDEX   **
24         **               and use it as you would INDEX.   **
25         *****
26           0035     NSYMS .set INDEX
27 000a 0035     .word NSYMS

```

.space/.bes
Reserve Space
Syntax

```
[label] .space size in bytes
```

```
[label] .bes size in bytes
```

Description

The **.space** and **.bes** directives reserve the number of bytes given by *size in bytes* in the current section and fill them with 0s. The section program counter is incremented to point to the word following the reserved space.

When you use a label with the **.space** directive, it points to the *first* byte reserved. When you use a label with the **.bes** directive, it points to the *last* byte reserved.

Example

This example shows how memory is reserved with the **.space** and **.bes** directives.

```

1          *****
2          ** Begin assembling into the .text section. **
3          *****
4 0000          .text
5
6          *****
7          ** Reserve 0F0 bytes in the .text section. **
8          *****
9 0000          .space 0F0h
10 00f0 0100     .word 100h, 200h
11             00f2 0200
12          *****
13          ** Begin assembling into the .data section. **
14          *****
15 0000          .data
16             .string "In .data"
17             0001 006E
18             0002 0020
19             0003 002E
20             0004 0064
21             0005 0061
22             0006 0074
23             0007 0061
24          *****
25          ** Reserve 100 bytes in the .data section; RES_1 **
26          ** points to the first byte that contains **
27          ** reserved bytes. **
28          *****
29 RES_1: .space 100
30             .word 15
31             .word RES_1
32          *****
33          ** Reserve 20 bits in the .data section; RES_2 **
34          ** points to the last byte that contains **
35          ** reserved bytes. **
36          *****
37 RES_2: .bes 20
38             .word 36h
39             .word RES_2

```

.sslist/.ssnolist **Control Listing of Substitution Symbols**

Syntax
.sslist
.ssnolist
Description

Two directives allow you to control substitution symbol expansion in the listing file:

The **.sslist** directive allows substitution symbol expansion in the listing file. The expanded line appears below the actual source line.

The **.ssnolist** directive suppresses substitution symbol expansion in the listing file.

By default, all substitution symbol expansion in the listing file is suppressed; the assembler acts as if the **.ssnolist** directive had been used.

Lines with the pound (#) character denote expanded substitution symbols.

Example

This example shows code that, by default, suppresses the listing of substitution symbol expansion, and it shows the **.sslist** directive assembled, instructing the assembler to list substitution symbol code expansion.

```

1          SHIFT .macro  dst,amount
2          .loop   amount
3          RLA    dst
4          .endloop
5          .endm
6
7          .global value
8
9 0000          SHIFT  R5,3
1         .loop   3
1         RLA    dst
1         .endloop
2         0000 5505  RLA    R5
2         0002 5505  RLA    R5
2         0004 5505  RLA    R5
10        0006 5582  ADD    R5,&value
          0008 0000!
11
12          .sslist
13
14 000a          SHIFT  R5,3
1         .loop   amount
#         .loop   3
1         RLA    dst
1         .endloop
2         000a 5505  RLA    dst
#         RLA    R5
2         000c 5505  RLA    dst
#         RLA    R5
2         000e 5505  RLA    dst
#         RLA    R5

```

.string
Initialize Text
Syntax

```
.string {expr1 | "string1"}, ... , {exprn | "stringn"}
```

Description

The **.string** directive places 8-bit characters from a character string into the current section. The *expr* or *string* can be one of the following:

- An expression that the assembler evaluates and treats as an 8-bit signed number.
- A character string enclosed in double quotes. Each character in a string represents a separate value, and values are stored in consecutive bytes. The entire string *must* be enclosed in quotes.
- A character string enclosed in double quotes. Each character in a string represents a separate byte. The entire string *must* be enclosed in quotes.

The assembler truncates any values that are greater than eight bits. You can have up to 100 operands, but they must fit on a single source statement line.

If you use a label, it points to the location of the first byte that is initialized.

When you use **.string** in a **.struct/.endstruct** sequence, **.stringdefine** a member's size; it does not initialize memory. For more information, see the [.struct/.endstruct/.tag](#) topic .

Example

In this example, 8-bit values are placed into consecutive words in the current section.

```

1 0000 0041  Str_Ptr:  .string  "ABCD"
   0001 0042
   0002 0043
   0003 0044
2 0004 0041                .string  41h, 42h, 43h, 44h
   0005 0042
   0006 0043
   0007 0044
3 0008 0041                .string  "Austin", "Houston", "Dallas"
   0009 0075
   000a 0073
   000b 0074
   000c 0069
   000d 006E
   000e 0048
   000f 006F
   0010 0075
   0011 0073
   0012 0074
   0013 006F
   0014 006E
   0015 0044
   0016 0061
   0017 006C
   0018 006C
   0019 0061
   001a 0073
4 001b 0030                .string  36 + 12
```

.struct/.endstruct/.tag Declare Structure Type

Syntax

```

[stag]  .struct  [expr]
[mem0] element [expr0]
[mem1] element [expr1]
.       .       .
.       .       .
.       .       .
[memn] .tag stag [exprn]
.       .       .
.       .       .
[memN] element [exprN]
[size]  .endstruct
label  .tag      stag

```

Description

The **.struct** directive assigns symbolic offsets to the elements of a data structure definition. This allows you to group similar data elements together and let the assembler calculate the element offset. This is similar to a C structure or a Pascal record. The **.struct** directive does not allocate memory; it merely creates a symbolic template that can be used repeatedly.

The **.endstruct** directive terminates the structure definition.

The **.tag** directive gives structure characteristics to a *label*, simplifying the symbolic representation and providing the ability to define structures that contain other structures. The **.tag** directive does not allocate memory. The structure tag (*stag*) of a **.tag** directive must have been previously defined.

Following are descriptions of the parameters used with the **.struct**, **.endstruct**, and **.tag** directives:

- The *stag* is the structure's tag. Its value is associated with the beginning of the structure. If no *stag* is present, the assembler puts the structure members in the global symbol table with the value of their absolute offset from the top of the structure. A **.stag** is optional for **.struct**, but is required for **.tag**.
- The *expr* is an optional expression indicating the beginning offset of the structure. The default starting point for a structure is 0.
- The *mem_{n/N}* is an optional label for a member of the structure. This label is absolute and equates to the present offset from the beginning of the structure. A label for a structure member cannot be declared global.
- The *element* is one of the following descriptors: **.byte**, **.char**, **.word**, **.int**, **.long**, **.string**, **.double**, **.float**, **.half**, **.short**, **.field**, and **.tag**. All of these except **.tag** are typical directives that initialize memory. Following a **.struct** directive, these directives describe the structure element's size. They do not allocate memory. The **.tag** directive is a special case because *stag* must be used (as in the definition of *stag*).
- The *expr_{n/N}* is an optional expression for the number of elements described. This value defaults to 1. A **.string** element is considered to be one byte in size, and a **.field** element is one bit.
- The *size* is an optional label for the total size of the structure.

Directives That Can Appear in a .struct/.endstruct Sequence

Note: The only directives that can appear in a **.struct/.endstruct** sequence are element descriptors, conditional assembly directives, and the **.align** directive, which aligns the member offsets on word boundaries. Empty structures are illegal.

The following examples show various uses of the .struct, .tag, and .endstruct directives.

Example 1

```

1          REAL_REC  .struct                ;stag
2      0000  NOM      .int                  ;member1 = 0
3      0002  DEN      .int                  ;member2 = 2
4      0004  REAL_LEN .endstruct           ;real_len = 4
5
6 0000          .bss    REAL, REAL_LEN
7
8 0000          .text
9 0000 521B     ADD.W   &REAL + REAL_REC.DEN,R11
      0002 0002!
10

```

Example 2

```

11 0000          .data
12          CPLX_REC .struct
13      0000  REALI   .tag    REAL_REC      ; stag
14      0004  IMAGI   .tag    REAL_REC      ; member1 = 0
15      0008  CPLX_LEN .endstruct          ; cplx_len = 8
16
17          COMPLEX .tag    CPLX_REC        ; assign structure attrib
18
19 0004          .bss    COMPLEX, CPLX_LEN
20
21 0004          .text
22 0004 521B     ADD    &COMPLEX.REALI,R11 ; access structure
      0006 0004!

```

Example 3

```

1 0000          .data
2          .struct                ; no stag puts mems into
3                                     ; global symbol table
4      0000  X        .int
5      0002  Y        .int
6      0004  Z        .int
7      0006          .endstruct

```

Example 4

```

1 0000          .data
2          BIT_REC  .struct                ; stag
3      0000  STREAM  .string 64
4      0040  BIT7    .field 7              ; bits1 = 64
5      0040  BIT9    .field 9              ; bits2 = 64
6      0042  BIT10   .field 10             ; bits3 = 65
7      0044  X_INT   .int                  ; x_int = 66
8      0046  BIT_LEN .endstruct           ; length = 67
9
10          BITS    .tag    BIT_REC
11
12 0000          .bss    BITS, BIT_REC
13
14 0000          .text
15 0000 521B     ADD    &BITS.BIT7,R11     ; move into R11
      0002 0040!
16 0004 F03B     AND    #127,R11          ; mask off garbage bits
      0006 007F

```


.tab
Define Tab Size
Syntax
.tab *size*
Description

The **.tab** directive defines the tab size. Tabs encountered in the source input are translated to *size* character spaces in the listing. The default tab size is eight spaces.

Example

In this example, each of the lines of code following a **.tab** statement consists of a single tab character followed by an NOP instruction.

Source file:

```

; default tab size
NOP
NOP
NOP
    .tab 4
NOP
NOP
NOP
    .tab 16
NOP
NOP
NOP
  
```

Listing file:

```

1          ; default tab size
2 0000 4303      NOP
3 0002 4303      NOP
4 0004 4303      NOP
5
7 0006 4303      NOP
8 0008 4303      NOP
9 000a 4303      NOP
10
12 000c 4303      NOP
13 000e 4303      NOP
14 0010 4303      NOP
  
```

.text *Assemble Into the .text Section*

Syntax
.text
Description

The **.text** directive tells the assembler to begin assembling into the **.text** section, which usually contains executable code. The section program counter is set to 0 if nothing has yet been assembled into the **.text** section. If code has already been assembled into the **.text** section, the section program counter is restored to its previous value in the section.

The **.text** section is the default section. Therefore, at the beginning of an assembly, the assembler assembles code into the **.text** section unless you use a **.data** or **.sect** directive to specify a different section.

For more information about sections, see [Chapter 2](#).

Example

This example assembles code into the **.text** and **.data** sections.

```

1          *****
2          * Begin assembling into the .data section. *
3          *****
4 0000          .data
5 0000 000A          .byte 0Ah, 0Bh
   0001 000B
6 0002 0011  coeff   .word 011h,0x22,0x33
   0004 0022
   0006 0033
7
8          *****
9          * Begin assembling into the .text section. *
10         *****
11 0000          .text
12 0000 0041  START: .string "A", "B", "C"
   0001 0042
   0002 0043
13 0003 0058  $END:  .string "X", "Y", "Z"
   0004 0059
   0005 005A
14 0006 403A          MOV.W   #0x1234,R10
   0008 1234
15 000a 521A          ADD.W   &coeff+1,R10
   000c 0003!
16
17         *****
18         * Resume assembling into .data section. *
19         *****
20 0008          .data
21 0008 000C          .byte  0Ch, 0Dh
   0009 000D
22
23         *****
24         * Resume assembling into .text section. *
25         *****
26 000e          .text
27 000e 0051          .string "QUIT"
   000f 0055
   0010 0049
   0011 0054

```

.title
Define Page Title
Syntax
.title "string"
Description

The **.title** directive supplies a title that is printed in the heading on each listing page. The source statement itself is not printed, but the line counter is incremented.

The *string* is a quote-enclosed title of up to 64 characters. If you supply more than 64 characters, the assembler truncates the string and issues a warning:

```
*** WARNING! line x: W0001: String is too long - will be truncated
```

The assembler prints the title on the page that follows the directive and on subsequent pages until another **.title** directive is processed. If you want a title on the first page, the first source statement must contain a **.title** directive.

Example

In this example, one title is printed on the first page and a different title is printed on succeeding pages.

Source file:

```
.title "**** Fast Fourier Transforms ****"
;
;
;
.title "**** Floating-Point Routines ****"
.page
```

Listing file:

```
MSP430 COFF Assembler PC vx.x.x      Day   Time   Year

Tools Copyright (c) 2003-2004 Texas Instruments Incorporated
**** Fast Fourier Transforms ****                                PAGE 1

      2          ;          .
      3          ;          .
      4          ;          .

MSP430 COFF Assembler PC vx.x.x      Day   Time   Year

Tools Copyright (c) 2003-2004 Texas Instruments Incorporated
**** Floating-Point Routines ****                                PAGE 2
```

.usect
Reserve Uninitialized Space

Syntax

symbol **.usect** "*section name*", *size in bytes* [, *alignment*]

Description

The **.usect** directive reserves space for variables in an uninitialized, named section. This directive is similar to the **.bss** directive; both simply reserve space for data and that space has no contents. However, **.usect** defines additional sections that can be placed anywhere in memory, independently of the **.bss** section.

- The *symbol* points to the first location reserved by this invocation of the **.usect** directive. The symbol corresponds to the name of the variable for which you are reserving space.
- The *section name* is significant to 200 characters and must be enclosed in double quotes. This parameter names the uninitialized section. A section name can contain a subsection name in the form *section name:subsection name*.
- The *size in bytes* is an expression that defines the number of bytes that are reserved in *section name*.
- The *alignment* is an optional parameter that ensures that the space allocated to the symbol occurs on the specified boundary. The boundary indicates the size of the slot in bytes and can be set to a power of 2 between 2^0 and 2^{15} , inclusive. If the SPC is aligned at the specified boundary, it is not incremented.

Initialized sections directives (**.text**, **.data**, and **.sect**) end the current section and tell the assembler to begin assembling into another section. A **.usect** or **.bss** directive encountered in the current section is simply assembled, and assembly continues in the current section.

Variables that can be located contiguously in memory can be defined in the same specified section; to do so, repeat the **.usect** directive with the same section name and the subsequent symbol (variable name).

For more information about sections, see [Chapter 2](#).

Example

This example uses the **.usect** directive to define two uninitialized, named sections, **var1** and **var2**. The symbol **ptr** points to the first byte reserved in the **var1** section. The symbol **array** points to the first byte in a block of 100 bytes reserved in **var1**, and **dflag** points to the first byte in a block of 50 bytes in **var1**. The symbol **vec** points to the first byte reserved in the **var2** section.

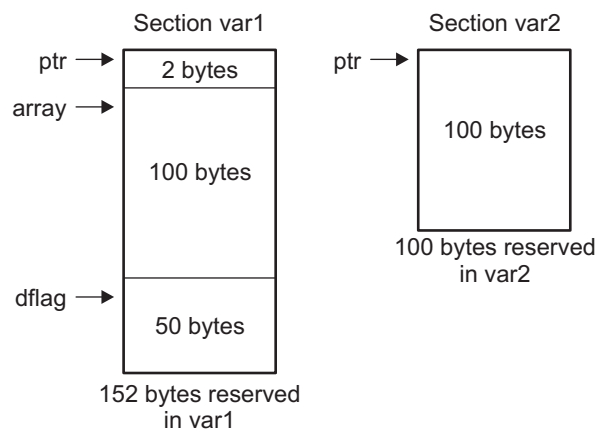
Figure 4-7 shows how this example reserves space in two uninitialized sections, var1 and var2.

```

1          *****
2          **          Assemble into the .text section.          **
3          *****
4 0000          .text
5 0000 403B          MOV #0x3,R11
   0002 0003
6
7          *****
8          **          Reserve 2 bytes in the var1 section.          **
9          *****
10 0000          ptr          .usect "var1",2
11
12          *****
13          **          Reserve 100 bytes in the var1 section.          **
14          *****
15 0002          array          .usect "var1",100
16
17 0004 503B          ADD #37,R11
   0006 0025
18
19          *****
20          **          Reserve 50 bytes in the var1 section.          **
21          *****
22 0066          dflag          .usect "var1",50
23
24 0008 503C          ADD #dflag-array,R12
   000a 0064
25
26          *****
27          ** Reserve 100 bytes in the var2 section. **
28          *****
29 0000          vec          .usect "var2",100
30
31 000c 5C0B          ADD R12,R11
32
33          *****
34          **          Declare a .usect symbol to be external.          **
35          *****
36          .global array

```

Figure 4-7. The .usect Directive



.var***Use Substitution Symbols as Local Variables***

Syntax

```
.var sym1[, sym2, ... , symn]
```

Description

The `.var` directive allows you to use substitution symbols as local variables within a macro. With this directive, you can define up to 32 local macro substitution symbols (including parameters) per macro.

The `.var` directive creates temporary substitution symbols with the initial value of the null string. These symbols are not passed in as parameters, and they are lost after expansion.

See [Chapter 5](#) for information on macros.

Macro Description

The MSP430™ assembler supports a macro language that enables you to create your own instructions. This is especially useful when a program executes a particular task several times. The macro language lets you:

- Define your own macros and redefine existing macros
- Simplify long or complicated assembly code
- Access macro libraries created with the archiver
- Define conditional and repeatable blocks within a macro
- Manipulate strings within a macro
- Control expansion listing

| Topic | Page |
|--|------------|
| 5.1 Using Macros | 120 |
| 5.2 Defining Macros | 120 |
| 5.3 Macro Parameters/Substitution Symbols | 122 |
| 5.4 Macro Libraries | 128 |
| 5.5 Using Conditional Assembly in Macros | 129 |
| 5.6 Using Labels in Macros | 131 |
| 5.7 Producing Messages in Macros | 132 |
| 5.8 Using Directives to Format the Output Listing | 133 |
| 5.9 Using Recursive and Nested Macros | 134 |
| 5.10 Macro Directives Summary | 135 |

5.1 Using Macros

Programs often contain routines that are executed several times. Instead of repeating the source statements for a routine, you can define the routine as a macro, then call the macro in the places where you would normally repeat the routine. This simplifies and shortens your source program.

If you want to call a macro several times but with different data each time, you can assign parameters within a macro. This enables you to pass different information to the macro each time you call it. The macro language supports a special symbol called a *substitution symbol*, which is used for macro parameters. See [Section 5.3](#) for more information.

Using a macro is a 3-step process.

- Step 1. **Define the macro.** You must define macros before you can use them in your program. There are two methods for defining macros:
 - a. Macros can be defined at the beginning of a *source file* or in a copy/include file. See [Section 5.2, Defining Macros](#), for more information.
 - b. Macros can also be defined in a *macro library*. A macro library is a collection of files in archive format created by the archiver. Each member of the archive file (macro library) may contain one macro definition corresponding to the member name. You can access a macro library by using the `.mlib` directive. For more information, see [Section 5.4](#).
- Step 2. **Call the macro.** After you have defined a macro, call it by using the macro name as a mnemonic in the source program. This is referred to as a *macro call*.
- Step 3. **Expand the macro.** The assembler expands your macros when the source program calls them. During expansion, the assembler passes arguments by variable to the macro parameters, replaces the macro call statement with the macro definition, then assembles the source code. By default, the macro expansions are printed in the listing file. You can turn off expansion listing by using the `.mnlst` directive. For more information, see [Section 5.8](#).

When the assembler encounters a macro definition, it places the macro name in the opcode table. This redefines any previously defined macro, library entry, directive, or instruction mnemonic that has the same name as the macro. This allows you to expand the functions of directives and instructions, as well as to add new instructions.

5.2 Defining Macros

You can define a macro anywhere in your program, but you must define the macro before you can use it. Macros can be defined at the beginning of a source file or in a `.copy/.include` file (see [Copy Source File](#)); they can also be defined in a macro library. For more information about macro libraries, see [Section 5.4](#).

Macro definitions can be nested, and they can call other macros, but all elements of the macro must be defined in the same file. Nested macros are discussed in [Section 5.9](#).

A macro definition is a series of source statements in the following format:

```

macname  .macro  [parameter1] [, ... ,parametern]
           model statements or macro directives
           [.mexit]
           .endm
  
```

macname names the macro. You must place the name in the source statement's label field. Only the first 128 characters of a macro name are significant. The assembler places the macro name in the internal opcode table, replacing any instruction or previous macro definition with the same name.

.macro is the directive that identifies the source statement as the first line of a macro definition. You must place `.macro` in the opcode field.

| | |
|--|--|
| <i>parameter</i> ₁ , <i>parameter</i> _n | are optional substitution symbols that appear as operands for the <code>.macro</code> directive. Parameters are discussed in Section 5.3 . |
| <i>model statements</i> | are instructions or assembler directives that are executed each time the macro is called. |
| <i>macro directives</i> | are used to control macro expansion. |
| .mexit | is a directive that functions as a <i>goto .endm</i> . The <code>.mexit</code> directive is useful when error testing confirms that macro expansion fails and completing the rest of the macro is unnecessary. |
| .endm | is the directive that terminates the macro definition. |

If you want to include comments with your macro definition but *do not* want those comments to appear in the macro expansion, use an exclamation point to precede your comments. If you *do* want your comments to appear in the macro expansion, use an asterisk or semicolon. See [Section 5.7](#) for more information about macro comments.

[Example 5-1](#) shows the definition, call, and expansion of a macro.

Example 5-1. Macro Definition, Call, and Expansion

Macro definition: The following code defines a macro, `add3`, with four parameters:

```

1          add3      .macro  P1,P2,P3,SUM
2                      MOV   #0,SUM
3                      ADD   P1,SUM
4                      ADD   P2,SUM
5                      ADD   P3,SUM
6                      .endm

```

Macro call: The following code calls the `add3` macro with four arguments:

```

7
8 0000          add3      R11,R12,R13,R14

```

Macro expansion: The following code shows the substitution of the macro definition for the macro call. The assembler substitutes `R11`, `R12`, `R13`, and `R14` for the `P1`, `P2`, `P3`, and `SUM` parameters of `add3`.

```

1          0000 430E      MOV   #0,R14
1          0002 5B0E      ADD   R11,R14
1          0004 5C0E      ADD   R12,R14
1          0006 5D0E      ADD   R13,R14

```

5.3 Macro Parameters/Substitution Symbols

If you want to call a macro several times with different data each time, you can assign parameters within the macro. The macro language supports a special symbol, called a *substitution symbol*, which is used for macro parameters.

Macro parameters are substitution symbols that represent a character string. These symbols can also be used outside of macros to equate a character string to a symbol name (see [Section 3.8.6](#)).

Valid substitution symbols can be up to 128 characters long and *must begin with a letter*. The remainder of the symbol can be a combination of alphanumeric characters, underscores, and dollar signs.

Substitution symbols used as macro parameters are local to the macro they are defined in. You can define up to 32 local substitution symbols (including substitution symbols defined with the `.var` directive) per macro. For more information about the `.var` directive, see [Section 5.3.6](#).

During macro expansion, the assembler passes arguments by variable to the macro parameters. The character-string equivalent of each argument is assigned to the corresponding parameter. Parameters without corresponding arguments are set to the null string. If the number of arguments exceeds the number of parameters, the last parameter is assigned the character-string equivalent of all remaining arguments.

If you pass a list of arguments to one parameter or if you pass a comma or semicolon to a parameter, you must surround these terms with quotation marks .

At assembly time, the assembler replaces the macro parameter/substitution symbol with its corresponding character string, then translates the source code into object code.

[Example 5-2](#) shows the expansion of a macro with varying numbers of arguments.

Example 5-2. Calling a Macro With Varying Numbers of Arguments

| | |
|---|--|
| <p>Macro definition:</p> <pre> Parms .macro a,b,c ; a = :a: ; b = :b: ; c = :c: .endm </pre> | |
| <p>Calling the macro:</p> | |
| <pre> Parms 100,label ; a = 100 ; b = label ; c = " " </pre> | <pre> Parms 100,label,x,y ; a = 100 ; b = label ; c = x,y </pre> |
| <pre> Parms 100, , x ; a = 100 ; b = " " ; c = x </pre> | <pre> Parms "100,200,300",x,y ; a = 100,200,300 ; b = x ; c = y </pre> |
| <pre> Parms ""string"",x,y ; a = "string" ; b = x ; c = y </pre> | |

5.3.1 Directives That Define Substitution Symbols

You can manipulate substitution symbols with the **.asg** and **.eval** directives.

- The **.asg** directive assigns a character string to a substitution symbol.
For the **.asg** directive, the quotation marks are optional. If there are no quotation marks, the assembler reads characters up to the first comma and removes leading and trailing blanks. In either case, a character string is read and assigned to the *substitution symbol*. The syntax of the **.asg** directive is:

```
.asg["]character string["], substitution symbol
```

[Example 5-3](#) shows character strings being assigned to substitution symbols.

Example 5-3. The **.asg** Directive

```
.asg R13, stack_ptr ; stack pointer
```

- The **.eval** directive performs arithmetic on numeric substitution symbols.
The **.eval** directive evaluates the *expression* and assigns the string value of the result to the *substitution symbol*. If the expression is not well defined, the assembler generates an error and assigns the null string to the symbol. The syntax of the **.eval** directive is:

```
.eval well-defined expression, substitution symbol
```

[Example 5-4](#) shows arithmetic being performed on substitution symbols.

Example 5-4. The **.eval** Directive

```
.asg 1,counter
.loop 100
.word counter
.eval counter + 1,counter
.endloop
```

In [Example 5-4](#), the **.asg** directive could be replaced with the **.eval** directive (**.eval 1, counter**) without changing the output. In simple cases like this, you can use **.eval** and **.asg** interchangeably. However, you must use **.eval** if you want to calculate a *value* from an expression. While **.asg** only assigns a character string to a substitution symbol, **.eval** evaluates an expression and then assigns the character string equivalent to a substitution symbol.

See [Assign a Substitution Symbol](#) for more information about the **.asg** and **.eval** assembler directives.

5.3.2 Built-In Substitution Symbol Functions

The following built-in substitution symbol functions enable you to make decisions on the basis of the string value of substitution symbols. These functions always return a value, and they can be used in expressions. Built-in substitution symbol functions are especially useful in conditional assembly expressions. Parameters of these functions are substitution symbols or character-string constants.

In the function definitions shown in [Table 5-1](#), *a* and *b* are parameters that represent substitution symbols or character-string constants. The term *string* refers to the string value of the parameter. The symbol *ch* represents a character constant.

Table 5-1. Substitution Symbol Functions and Return Values

| Function | Return Value |
|--|--|
| \$symlen (<i>a</i>) | Length of string <i>a</i> |
| \$symcmp (<i>a,b</i>) | < 0 if <i>a</i> < <i>b</i> ; 0 if <i>a</i> = <i>b</i> ; > 0 if <i>a</i> > <i>b</i> |
| \$firstch (<i>a,ch</i>) | Index of the first occurrence of character constant <i>ch</i> in string <i>a</i> |
| \$lastch (<i>a,ch</i>) | Index of the last occurrence of character constant <i>ch</i> in string <i>a</i> |
| \$isdefed (<i>a</i>) | 1 if string <i>a</i> is defined in the symbol table 0 if string <i>a</i> is not defined in the symbol table |
| \$ismember (<i>a,b</i>) | Top member of list <i>b</i> is assigned to string <i>a</i> 0 if <i>b</i> is a null string |
| \$iscons (<i>a</i>) | 1 if string <i>a</i> is a binary constant 2 if string <i>a</i> is an octal constant 3 if string <i>a</i> is a hexadecimal constant 4 if string <i>a</i> is a character constant 5 if string <i>a</i> is a decimal constant |
| \$isname (<i>a</i>) | 1 if string <i>a</i> is a valid symbol name 0 if string <i>a</i> is not a valid symbol name |
| \$isreg (<i>a</i>) ⁽¹⁾ | 1 if string <i>a</i> is a valid predefined register name 0 if string <i>a</i> is not a valid predefined register name |

⁽¹⁾ For more information about predefined register names, see [Section 3.8.5](#).

[Example 5-5](#) shows built-in substitution symbol functions.

Example 5-5. Using Built-In Substitution Symbol Functions

```
.asg  label, ADDR          ; ADDR = label
.if   ($symcmp(ADDR, "label") = 0) ; evaluates to true
MOV   #ADDR, R4
.endif
.asg  "x,y,z" , list      ; list = x,y,z
.if   ($ismember(ADDR,list)) ; ADDR = x, list = y,z
SUB   #4, R4              ; sub x
.endif
```

5.3.3 Recursive Substitution Symbols

When the assembler encounters a substitution symbol, it attempts to substitute the corresponding character string. If that string is also a substitution symbol, the assembler performs substitution again. The assembler continues doing this until it encounters a token that is not a substitution symbol or until it encounters a substitution symbol that it has already encountered during this evaluation.

In [Example 5-6](#), the x is substituted for z; z is substituted for y; and y is substituted for x. The assembler recognizes this as infinite recursion and ceases substitution.

Example 5-6. Recursive Substitution

```
.asg "x",z ; declare z and assign z = "x"
.asg "z",y ; declare y and assign y = "z"
.asg "y",x ; declare x and assign x = "y"
MOV #x, R11
```

5.3.4 Forced Substitution

In some cases, substitution symbols are not recognizable to the assembler. The forced substitution operator, which is a set of colons surrounding the symbol, enables you to force the substitution of a symbol's character string. Simply enclose a symbol with colons to force the substitution. Do not include any spaces between the colons and the symbol.

The syntax for the forced substitution operator is:

```
:symbol:
```

The assembler expands substitution symbols surrounded by colons before expanding other substitution symbols.

You can use the forced substitution operator only inside macros, and you cannot nest a forced substitution operator within another forced substitution operator.

[Example 5-7](#) shows how the forced substitution operator is used.

Example 5-7. Using the Forced Substitution Operator

```

1          force .macro
2          .asg 0,x
3          .loop 8
4          AUX:x: .set x
5          .eval x+1,x
6          .endloop
7          .endm
8
9 0000          force
1         .asg 0,x
1         .loop 8
1         AUX:x: .set x
1         .eval x+1,x
1         .endloop
2         0000 AUX0 .set 0
2         .eval 0+1,x
2         0001 AUX1 .set 1
2         .eval 1+1,x
2         0002 AUX2 .set 2
2         .eval 2+1,x
2         0003 AUX3 .set 3
2         .eval 3+1,x
2         0004 AUX4 .set 4
2         .eval 4+1,x
2         0005 AUX5 .set 5
2         .eval 5+1,x
2         0006 AUX6 .set 6
2         .eval 6+1,x
2         0007 AUX7 .set 7
2         .eval 7+1,x

```

5.3.5 Accessing Individual Characters of Subscripted Substitution Symbols

In a macro, you can access the individual characters (substrings) of a substitution symbol with subscripted substitution symbols. You must use the forced substitution operator for clarity.

You can access substrings in two ways:

- `:symbol (well-defined expression)`:
This method of subscripting evaluates to a character string with one character.
- `:symbol (well-defined expression1, well-defined expression2)`:
In this method, expression₁ represents the substring's starting position, and expression₂ represents the substring's length. You can specify exactly where to begin subscripting and the exact length of the resulting character string. *The index of substring characters begins with 1, not 0.*

[Example 5-8](#) and [Example 5-9](#) show built-in substitution symbol functions used with subscripted substitution symbols.

In [Example 5-8](#), subscripted substitution symbols redefine the ADD instruction so that it handles short immediate values. In [Example 5-9](#), the subscripted substitution symbol is used to find a substring strg1 beginning at position start in the string strg2. The position of the substring strg1 is assigned to the substitution symbol pos.

Example 5-8. Using Subscripted Substitution Symbols to Redefine an Instruction

```

ADDX    .macro      dst, imm
        .var        TMP
        .asg        :imm(1):, TMP
        .if         $symcmp(TMP,"#") = 0
        ADD         imm, dst
        .else
        .emsg       "Bad Macro Parameter"
        .endif
        .endm

ADDX    R9, #100          ; macro call
ADDX    R9, R8           ; macro call
    
```

Example 5-9. Using Subscripted Substitution Symbols to Find Substrings

```

substr  .macro      start, strg1, strg2, pos
        .var        LEN1, LEN2, I, TMP
        .if         $symlen(start) = 0
        .eval      1, start
        .endif
        .eval      0, pos
        .eval      1, i
        .eval      $symlen(strg1), LEN1
        .eval      $symlen(strg2), LEN2
        .loop
        .break     i = (LEN2 - LEN1 + 1)
        .asg       ":strg2(I,LEN1):", TMP
        .eval      i, pos
        .break
        .else
        .eval      i + 1, i
        .endif
        .endloop
        .endm

        .asg       0, pos
        .asg       "ar1 ar2 ar3 ar4", regs
        substr     1, "ar2", regs, pos
        .word      pos
    
```

5.3.6 Substitution Symbols as Local Variables in Macros

If you want to use substitution symbols as local variables within a macro, you can use the `.var` directive to define up to 32 local macro substitution symbols (including parameters) per macro. The `.var` directive creates temporary substitution symbols with the initial value of the null string. These symbols are not passed in as parameters, and they are lost after expansion.

```
.var sym1 [,sym2 , ... ,symn]
```

The `.var` directive is used in [Example 5-8](#) and [Example 5-9](#).

5.4 Macro Libraries

One way to define macros is by creating a macro library. A macro library is a collection of files that contain macro definitions. You must use the archiver to collect these files, or members, into a single file (called an archive). Each member of a macro library contains one macro definition. The files in a macro library must be unassembled source files. The macro name and the member name must be the same, and the macro filename's extension must be .asm. For example:

| Macro Name | Filename in Macro Library |
|------------|---------------------------|
| simple | simple.asm |
| add3 | add3.asm |

You can access the macro library by using the `.mlib` assembler directive (described in [Define Macro Library](#)). The syntax is:

```
.mlib filename
```

When the assembler encounters the `.mlib` directive, it opens the library named by filename and creates a table of the library's contents. The assembler enters the names of the individual members within the library into the opcode tables as library entries; this redefines any existing opcodes or macros that have the same name. If one of these macros is called, the assembler extracts the entry from the library and loads it into the macro table.

The assembler expands the library entry in the same way it expands other macros. See [Section 5.1](#) for how the assembler expands macros. You can control the listing of library entry expansions with the `.mlist` directive. For more information about the `.mlist` directive, see [Section 5.8](#) and [Start/Stop Macro Expansion Listing](#). Only macros that are actually called from the library are extracted, and they are extracted only once.

You can use the archiver to create a macro library by including the desired files in an archive. A macro library is no different from any other archive, except that the assembler expects the macro library to contain macro definitions. The assembler expects *only* macro definitions in a macro library; putting object code or miscellaneous source files into the library may produce undesirable results. For information about creating a macro library archive, see [Section 6.1](#).

5.5 Using Conditional Assembly in Macros

The conditional assembly directives are **.if/.elseif/.else/.endif** and **.loop/.break/.endloop**. They can be nested within each other up to 32 levels deep. The format of a conditional block is:

```
.if well-defined expression
[.elseif well-defined expression]
[.else]
.endif
```

The **.elseif** and **.else** directives are optional in conditional assembly. The **.elseif** directive can be used more than once within a conditional assembly code block. When **.elseif** and **.else** are omitted and when the **.if** expression is false (0), the assembler continues to the code following the **.endif** directive. See [Assemble Conditional Blocks](#) for more information on the **.if/.elseif/.else/.endif** directives.

The **.loop/.break/.endloop** directives enable you to assemble a code block repeatedly. The format of a repeatable block is:

```
.loop [well-defined expression]
[.break [well-defined expression]]
.endloop
```

The **.loop** directive's optional *well-defined expression* evaluates to the loop count (the number of loops to be performed). If the expression is omitted, the loop count defaults to 1024 unless the assembler encounters a **.break** directive with an expression that is true (nonzero). See [Assemble Conditional Blocks Repeatedly](#) for more information on the **.loop/.break/.endloop** directives.

The **.break** directive and its expression are optional in repetitive assembly. If the expression evaluates to false, the loop continues. The assembler breaks the loop when the **.break** expression evaluates to true or when the **.break** expression is omitted. When the loop is broken, the assembler continues with the code after the **.endloop** directive.

For more information, see [Section 4.7](#).

[Example 5-10](#), [Example 5-11](#), and [Example 5-12](#) show the **.loop/.break/.endloop** directives, properly nested conditional assembly directives, and built-in substitution symbol functions used in a conditional assembly code block.

Example 5-10. The .loop/.break/.endloop Directives

```
.asg    1,x
.loop

.break  (x == 10) ; if x == 10, quit loop/break with expression

.eval  x+1,x
.endloop
```

Example 5-11. Nested Conditional Assembly Directives

```

.asg    1,x
.loop

.if     (x == 10) ; if x == 10, quit loop
.break  (x == 10) ; force break
.endif

.eval  x+1,x
.endloop

```

Example 5-12. Built-In Substitution Symbol Functions in a Conditional Assembly Code Block

```

.fcncolist

* Increment or decrement
INCDEC .macro OP,dst
    .if     $symcmp(OP,"+")
        ADD    #1,dst
    .elseif $symcmp(OP,"-")
        SUB    #1,dst
    .else
        .emsg  "Incorrect Operator Parameter"
    .endif
.endm

* Macro Call
INCDEC  +,R11

```

5.6 Using Labels in Macros

All labels in an assembly language program must be unique. This includes labels in macros. If a macro is expanded more than once, its labels are defined more than once. *Defining a label more than once is illegal.* The macro language provides a method of defining labels in macros so that the labels are unique. Simply follow each label with a question mark, and the assembler replaces the question mark with a period followed by a unique number. When the macro is expanded, *you do not see the unique number in the listing file.* Your label appears with the question mark as it did in the macro definition. You cannot declare this label as global. The syntax for a unique label is:

```
label ?
```

[Example 5-13](#) shows unique label generation in a macro. The maximum label length is shortened to allow for the unique suffix. For example, if the macro is expanded fewer than 10 times, the maximum label length is 126 characters. If the macro is expanded from 10 to 99 times, the maximum label length is 125. The label with its unique suffix is shown in the cross-listing file. To obtain a cross-listing file, invoke the assembler with the `--cross_reference` option (see [Section 3.3](#)).

Example 5-13. Unique Labels in a Macro

```

1          * Define macro to find minimum
2
3          MIN      .macro  src1,src2,dst
4                  CMP      src1,src2
5                  JL       m1?
6                  MOV      src1,dst
7                  JMP      m2?
8          m1?     MOV      src2,dst
9          m2?
10         .endm
11
12 0000          MIN      R11,R12,R13
1 0000 9B0C      CMP      R11,R12
1 0002 3802      JL       m1?
1 0004 4B0D      MOV      R11,R13
1 0006 3C01      JMP      m2?
1 0008 4C0D      MOV      R12,R13
1                  m1?
1                  m2?

```

5.7 Producing Messages in Macros

The macro language supports three directives that enable you to define your own assembly-time error and warning messages. These directives are especially useful when you want to create messages specific to your needs. The last line of the listing file shows the error and warning counts. These counts alert you to problems in your code and are especially useful during debugging.

- .emsg** sends error messages to the listing file. The `.emsg` directive generates errors in the same manner as the assembler, incrementing the error count and preventing the assembler from producing an object file.
- .mmsg** sends assembly-time messages to the listing file. The `.mmsg` directive functions in the same manner as the `.emsg` directive but does not set the error count or prevent the creation of an object file.
- .wmsg** sends warning messages to the listing file. The `.wmsg` directive functions in the same manner as the `.emsg` directive, but it increments the warning count and does not prevent the generation of an object file.

Macro comments are comments that appear in the definition of the macro *but do not show up in the expansion of the macro*. An exclamation point in column 1 identifies a macro comment. If you want your comments to appear in the macro expansion, precede your comment with an asterisk or semicolon.

[Example 5-14](#) shows user messages in macros and macro comments that do not appear in the macro expansion.

For more information about the `.emsg`, `.mmsg`, and `.wmsg` assembler directives, see [Define Messages](#).

Example 5-14. Producing Messages in a Macro

```

1          MUL_I  .macro x,y
2              .if ($symlen(x) == 0)
3                  .emsg "ERROR -- Missing Parameter"
4                  .mexit
5                  .elseif ($symlen(y) == 0)
6                  .emsg "ERROR -- Missing Parameter"
7                  .mexit
8                  .else
9                      MOV  x, R11
10                     MOV  y, R12
11                 .endif
12                 .endm
13
14 0000          MUL_I #50, #51
1          .if ($symlen(x) == 0)
1              .emsg "ERROR -- Missing Parameter"
1              .mexit
1              .elseif ($symlen(y) == 0)
1              .emsg "ERROR -- Missing Parameter"
1              .mexit
1              .else
1                  MOV  #50, R11
1          0000 403B          MOV  #50, R11
1          0002 0032
1          0004 403C          MOV  #51, R12
1          0006 0033
1              .endif
1          15
16 0008          MUL_I
1          .if ($symlen(x) == 0)
1              .emsg "ERROR -- Missing Parameter"
"macromsg.asm", ERROR! at line 16: [ ***** USER ERROR ***** - ] ERROR -- Missing Parameter
1              .mexit
1          17
1          18

1 Assembly Error, No Assembly Warnings
    
```

5.8 Using Directives to Format the Output Listing

Macros, substitution symbols, and conditional assembly directives may hide information. You may need to see this hidden information, so the macro language supports an expanded listing capability.

By default, the assembler shows macro expansions and false conditional blocks in the list output file. You may want to turn this listing off or on within your listing file. Four sets of directives enable you to control the listing of this information:

- **Macro and loop expansion listing**

.mlist expands macros and `.loop/.endloop` blocks. The `.mlist` directive prints all code encountered in those blocks.

.mnolist suppresses the listing of macro expansions and `.loop/ .endloop` blocks.

For macro and loop expansion listing, `.mlist` is the default.

- **False conditional block listing**

.fclist causes the assembler to include in the listing file all conditional blocks that do not generate code (false conditional blocks). Conditional blocks appear in the listing exactly as they appear in the source code.

.fcnolist suppresses the listing of false conditional blocks. Only the code in conditional blocks that actually assemble appears in the listing. The `.if`, `.elseif`, `.else`, and `.endif` directives do not appear in the listing.

For false conditional block listing, `.fclist` is the default.

- **Substitution symbol expansion listing**

.sslist expands substitution symbols in the listing. This is useful for debugging the expansion of substitution symbols. The expanded line appears below the actual source line.

.ssnolist turns off substitution symbol expansion in the listing.

For substitution symbol expansion listing, `.ssnolist` is the default.

- **Directive listing**

.drlist causes the assembler to print to the listing file all directive lines.

.drnolist suppresses the printing of certain directives in the listing file. These directives are `.asg`, `.eval`, `.var`, `.sslist`, `.mlist`, `.fclist`, `.ssnolist`, `.mnolist`, `.fcnolist`, `.emsg`, `.wmsg`, `.mmsg`, `.length`, `.width`, and `.break`.

For directive listing, `.drlist` is the default.

5.9 Using Recursive and Nested Macros

The macro language supports recursive and nested macro calls. This means that you can call other macros in a macro definition. You can nest macros up to 32 levels deep. When you use recursive macros, you call a macro from its own definition (the macro calls itself).

When you create recursive or nested macros, you should pay close attention to the arguments that you pass to macro parameters because the assembler uses dynamic scoping for parameters. This means that the called macro uses the environment of the macro from which it was called.

[Example 5-15](#) shows nested macros. The `y` in the `in_block` macro hides the `y` in the `out_block` macro. The `x` and `z` from the `out_block` macro, however, are accessible to the `in_block` macro.

Example 5-15. Using Nested Macros

```

in_block .macro y,a
        .
        ; visible parameters are y,a and x,z from the calling macro
        .endm

out_block .macro x,y,z
        .
        ; visible parameters are x,y,z
        .
        in_block x,y ; macro call with x and y as arguments
        .
        .endm
out_block ; macro call

```

[Example 5-16](#) shows recursive and fact macros. The `fact` macro produces assembly code necessary to calculate the factorial of `n`, where `n` is an immediate value. The result is placed in data memory address `loc`. The `fact` macro accomplishes this by calling `fact1`, which calls itself recursively.

Example 5-16. Using Recursive Macros

```

fact .macro N,loc
    .if N < 2
    MOV #1,&loc
    .else
    MOV #N,&loc
    .eval N-1,N
    fact1
    .endif
    .endm

fact1 .macro
    .if N > 1
    MOV #N,R12 ; Assume MPY requires args to be in R12,R13
    MOV &loc,R13
    CALL MPY
    MOV R12,&loc ; Assume MPY returns product in R12
    .eval N - 1,N
    fact1
    .endif

    .endm

.global fact_result
.global MPY

fact 5,fact_result

```

5.10 Macro Directives Summary

The directives listed in [Table 5-2](#) through [Table 5-6](#) can be used with macros. The `.macro`, `.mexit`, `.endm` and `.var` directives are valid only with macros; the remaining directives are general assembly language directives.

Table 5-2. Creating Macros

| Mnemonic and Syntax | Description | See | |
|--|---|-----------------------------|-----------------------------|
| | | Macro Use | Directive |
| <code>.endm</code> | End macro definition | Section 5.2 | <code>.endm</code> |
| <code>macname .macro [parameter₁][... , parameter_n]</code> | Define macro by <i>macname</i> | Section 5.2 | <code>.macro</code> |
| <code>.mexit</code> | Go to <code>.endm</code> | Section 5.2 | Section 5.2 |
| <code>.mlib filename</code> | Identify library containing macro definitions | Section 5.4 | <code>.mlib</code> |

Table 5-3. Manipulating Substitution Symbols

| Mnemonic and Syntax | Description | See | |
|--|--|-------------------------------|--------------------|
| | | Macro Use | Directive |
| <code>.asg ["character string"], substitution symbol</code> | Assign character string to substitution symbol | Section 5.3.1 | <code>.asg</code> |
| <code>.eval well-defined expression, substitution symbol</code> | Perform arithmetic on numeric substitution symbols | Section 5.3.1 | <code>.eval</code> |
| <code>.var sym₁ [,sym₂ , ...,sym_n]</code> | Define local macro symbols | Section 5.3.6 | <code>.var</code> |

Table 5-4. Conditional Assembly

| Mnemonic and Syntax | Description | See | |
|---|-------------------------------------|-----------------------------|-----------------------|
| | | Macro Use | Directive |
| <code>.break [well-defined expression]</code> | Optional repeatable block assembly | Section 5.5 | <code>.break</code> |
| <code>.endif</code> | End conditional assembly | Section 5.5 | <code>.endif</code> |
| <code>.endloop</code> | End repeatable block assembly | Section 5.5 | <code>.endloop</code> |
| <code>.else</code> | Optional conditional assembly block | Section 5.5 | <code>.else</code> |
| <code>.elseif well-defined expression</code> | Optional conditional assembly block | Section 5.5 | <code>.elseif</code> |
| <code>.if well-defined expression</code> | Begin conditional assembly | Section 5.5 | <code>.if</code> |
| <code>.loop [well-defined expression]</code> | Begin repeatable block assembly | Section 5.5 | <code>.loop</code> |

Table 5-5. Producing Assembly-Time Messages

| Mnemonic and Syntax | Description | See | |
|---------------------|---|-----------------------------|--------------------|
| | | Macro Use | Directive |
| <code>.emsg</code> | Send error message to standard output | Section 5.7 | <code>.emsg</code> |
| <code>.mmsg</code> | Send assembly-time message to standard output | Section 5.7 | <code>.mmsg</code> |
| <code>.wmsg</code> | Send warning message to standard output | Section 5.7 | <code>.wmsg</code> |

Table 5-6. Formatting the Listing

| Mnemonic and Syntax | Description | See | |
|------------------------|---|-----------------------------|------------------------|
| | | Macro Use | Directive |
| <code>.fclist</code> | Allow false conditional code block listing (default) | Section 5.8 | <code>.fclist</code> |
| <code>.fcnolist</code> | Suppress false conditional code block listing | Section 5.8 | <code>.fcnolist</code> |
| <code>.mlist</code> | Allow macro listings (default) | Section 5.8 | <code>.mlist</code> |
| <code>.mno list</code> | Suppress macro listings | Section 5.8 | <code>.mno list</code> |
| <code>.sslist</code> | Allow expanded substitution symbol listing | Section 5.8 | <code>.sslist</code> |
| <code>.ssnolist</code> | Suppress expanded substitution symbol listing (default) | Section 5.8 | <code>.ssnolist</code> |

Archiver Description

The MSP430™ archiver lets you combine several individual files into a single archive file. For example, you can collect several macros into a macro library. The assembler searches the library and uses the members that are called as macros by the source file. You can also use the archiver to collect a group of object files into an object library. The linker includes in the library the members that resolve external references during the link. The archiver allows you to modify a library by deleting, replacing, extracting, or adding members.

| Topic | Page |
|---|------------|
| 6.1 Archiver Overview | 138 |
| 6.2 The Archiver's Role in the Software Development Flow | 139 |
| 6.3 Invoking the Archiver..... | 140 |
| 6.4 Archiver Examples | 141 |

6.1 Archiver Overview

You can build libraries from any type of files. Both the assembler and the linker accept archive libraries as input; the assembler can use libraries that contain individual source files, and the linker can use libraries that contain individual object files.

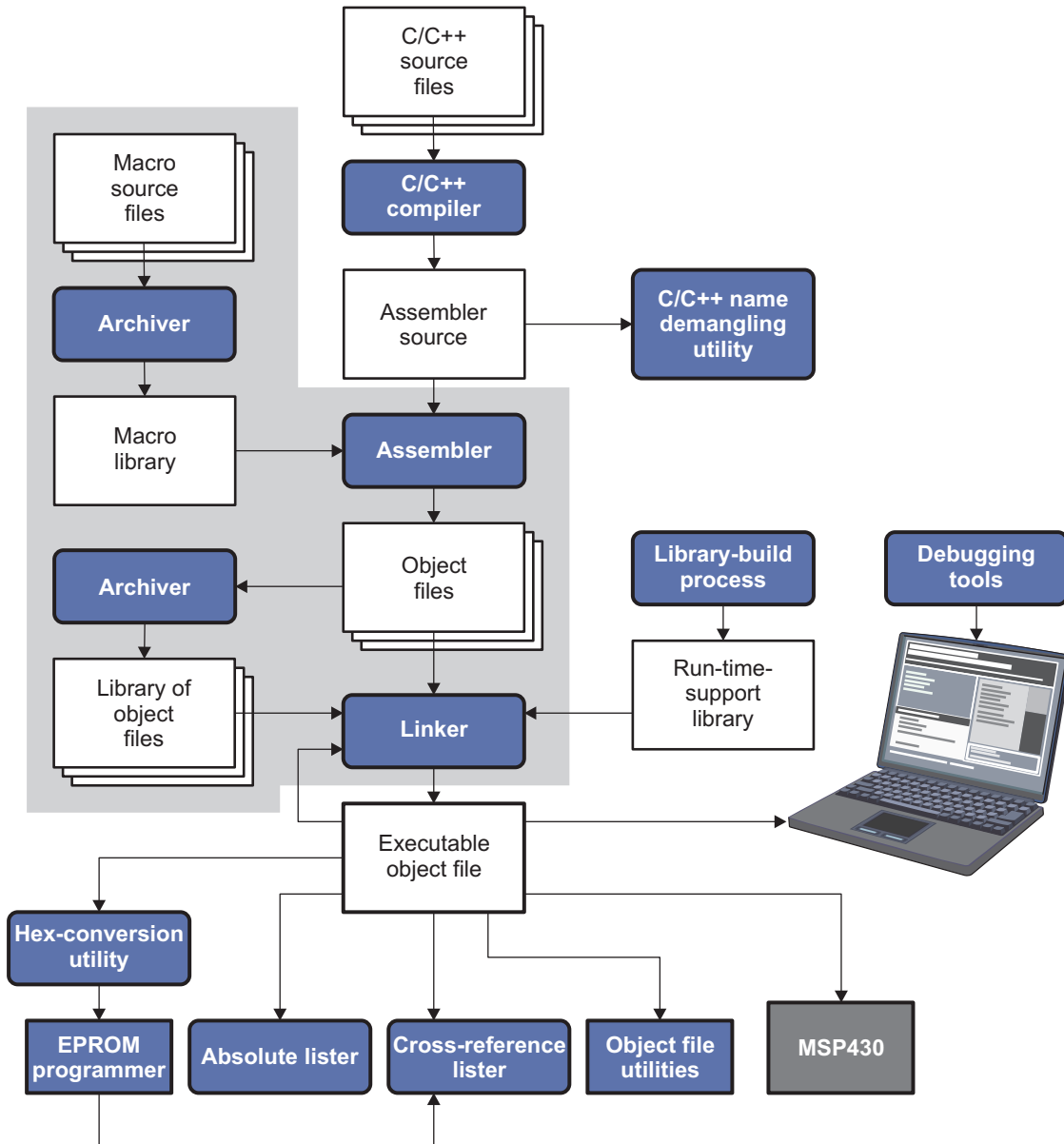
One of the most useful applications of the archiver is building libraries of object modules. For example, you can write several arithmetic routines, assemble them, and use the archiver to collect the object files into a single, logical group. You can then specify the object library as linker input. The linker searches the library and includes members that resolve external references.

You can also use the archiver to build macro libraries. You can create several source files, each of which contains a single macro, and use the archiver to collect these macros into a single, functional group. You can use the `.mlib` directive during assembly to specify that macro library to be searched for the macros that you call. [Chapter 5, *Macro Language*](#), discusses macros and macro libraries in detail, while this chapter explains how to use the archiver to build libraries.

6.2 The Archiver's Role in the Software Development Flow

Figure 6-1 shows the archiver's role in the software development process. The shaded portion highlights the most common archiver development path. Both the assembler and the linker accept libraries as input.

Figure 6-1. The Archiver in the MSP430 Software Development Flow



6.3 Invoking the Archiver

To invoke the archiver, enter:

```
ar430 [-]command [options] libname [filename1 ... filenamen]
```

ar430 is the command that invokes the archiver.

[-]command tells the archiver how to manipulate the existing library members and any specified . A command can be preceded by an optional hyphen. You must use one of the following commands when you invoke the archiver, but you can use only one command per invocation. The archiver commands are as follows:

- @** uses the contents of the specified file instead of command line entries. You can use this command to avoid limitations on command line length imposed by the host operating system. Use a ; at the beginning of a line in the command file to include comments. (See [Example 6-1](#) for an example using an archiver command file.)
- a** adds the specified files to the library. This command does not replace an existing member that has the same name as an added file; it simply *appends* new members to the end of the archive.
- d** deletes the specified members from the library.
- r** replaces the specified members in the library. If you do not specify filenames, the archiver replaces the library members with files of the same name in the current directory. If the specified file is not found in the library, the archiver adds it instead of replacing it.
- t** prints a table of contents of the library. If you specify filenames, only those files are listed. If you do not specify any filenames, the archiver lists all the members in the specified library.
- x** extracts the specified files. If you do not specify member names, the archiver extracts all library members. When the archiver extracts a member, it simply copies the member into the current directory; it *does not* remove it from the library.

options In addition to one of the *commands*, you can specify options. To use options, combine them with a command; for example, to use the a command and the s option, enter -as or as. The hyphen is optional for archiver options only. These are the archiver options:

- q** (quiet) suppresses the banner and status messages.
- s** prints a list of the global symbols that are defined in the library. (This option is valid only with the a, r, and d commands.)
- u** replaces library members only if the replacement has a more recent modification date. You must use the r command with the -u option to specify which members to replace.
- v** (verbose) provides a file-by-file description of the creation of a new library from an old library and its members.

libname names the archive library to be built or modified. If you do not specify an extension for *libname*, the archiver uses the default extension *.lib*.

filenames names individual files to be manipulated. These files can be existing library members or new files to be added to the library. When you enter a filename, you must enter a complete filename including extension, if applicable.

Naming Library Members

- Note:** It is possible (but not desirable) for a library to contain several members with the same name. If you attempt to delete, replace, or extract a member whose name is the same as another library member, the archiver deletes, replaces, or extracts the first library member with that name.

6.4 Archiver Examples

The following are examples of typical archiver operations:

- If you want to create a library called `function.lib` that contains the files `sine.obj`, `cos.obj`, and `flt.obj`, enter:

```
ar430 -a function sine.obj cos.obj flt.obj
```

The archiver responds as follows:

```
==> new archive 'function.lib'
==> building new archive 'function.lib'
```

- You can print a table of contents of `function.lib` with the `-t` command, enter:

```
ar430 -t function
```

The archiver responds as follows:

| FILE NAME | SIZE | DATE |
|-----------|------|--------------------------|
| sine.obj | 300 | Wed Jun 14 10:00:24 2006 |
| cos.obj | 300 | Wed Jun 14 10:00:30 2006 |
| flt.obj | 300 | Wed Jun 14 09:59:56 2006 |

- If you want to add new members to the library, enter:

```
ar430 -as function atan.obj
```

The archiver responds as follows:

```
==> symbol defined: '_sin'
==> symbol defined: '$sin'
==> symbol defined: '_cos'
==> symbol defined: '$cos'
==> symbol defined: '_tan'
==> symbol defined: '$tan'
==> symbol defined: '_atan'
==> symbol defined: '$atan'
==> building archive 'function.lib'
```

Because this example does not specify an extension for the libname, the archiver adds the files to the library called `function.lib`. If `function.lib` does not exist, the archiver creates it. (The `-s` option tells the archiver to list the global symbols that are defined in the library.)

- If you want to modify a library member, you can extract it, edit it, and replace it. In this example, assume there is a library named `macros.lib` that contains the members `push.asm`, `pop.asm`, and `swap.asm`.

```
ar430 -x macros push.asm
```

The archiver makes a copy of `push.asm` and places it in the current directory; it does not remove `push.asm` from the library. Now you can edit the extracted file. To replace the copy of `push.asm` in the library with the edited copy, enter:

```
ar430 -r macros push.asm
```

- If you want to use a command file, specify the command filename after the -@ command. For example:

```
ar430 -@modules.cmd
```

The archiver responds as follows:

```
==> building archive 'modules.lib'
```

Example 6-1 is the modules.cmd command file. The r command specifies that the filenames given in the command file replace files of the same name in the modules.lib library. The -u option specifies that these files are replaced only when the current file has a more recent revision date than the file that is in the library.

Example 6-1. Archiver Command File

```
; Command file to replace members of the
;   modules library with updated files
; Use r command and u option:
ru
; Specify library name:
modules.lib
; List filenames to be replaced if updated:
align.asm
bss.asm
data.asm
text.asm
sect.asm
clink.asm
copy.asm
double.asm
drnolist.asm
emsg.asm
end.asm
```

Linker Description

The MSP430™ linker creates executable modules by combining object modules. This chapter describes the linker options, directives, and statements used to create executable modules. Object libraries, command files, and other key concepts are discussed as well.

The concept of sections is basic to linker operation; [Chapter 2](#) discusses the object module sections in detail.

| Topic | Page |
|--|------|
| 7.1 Linker Overview | 144 |
| 7.2 The Linker's Role in the Software Development Flow | 145 |
| 7.3 Invoking the Linker | 146 |
| 7.4 Linker Options | 147 |
| 7.5 Linker Command Files | 164 |
| 7.6 Object Libraries | 166 |
| 7.7 The MEMORY Directive | 167 |
| 7.8 The SECTIONS Directive | 169 |
| 7.9 Specifying a Section's Run-Time Address | 181 |
| 7.10 Using UNION and GROUP Statements | 183 |
| 7.11 Special Section Types (DSECT, COPY, and NOLOAD) | 187 |
| 7.12 Default Allocation Algorithm | 187 |
| 7.13 Assigning Symbols at Link Time | 189 |
| 7.14 Creating and Filling Holes | 194 |
| 7.15 Linker-Generated Copy Tables | 197 |
| 7.16 Partial (Incremental) Linking | 205 |
| 7.17 Linking C/C++ Code | 206 |
| 7.18 Linker Example | 209 |

7.1 Linker Overview

The MSP430 linker allows you to configure system memory by allocating output sections efficiently into the memory map. As the linker combines object files, it performs the following tasks:

- Allocates sections into the target system's configured memory
- Relocates symbols and sections to assign them to final addresses
- Resolves undefined external references between input files

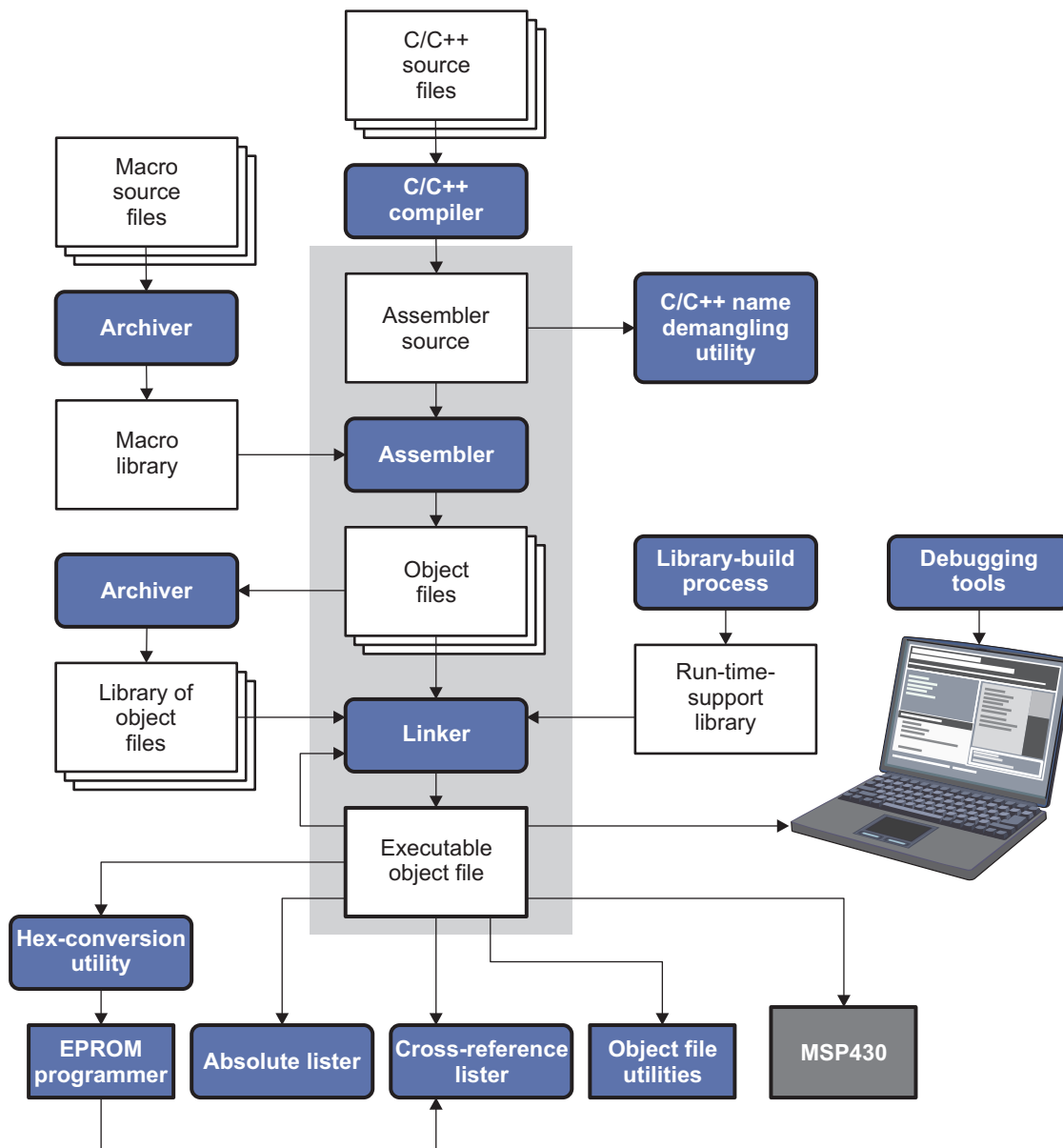
The linker command language controls memory configuration, output section definition, and address binding. The language supports expression assignment and evaluation. You configure system memory by defining and creating a memory model that you design. Two powerful directives, MEMORY and SECTIONS, allow you to:

- Allocate sections into specific areas of memory
- Combine object file sections
- Define or redefine global symbols at link time

7.2 The Linker's Role in the Software Development Flow

Figure 7-1 illustrates the linker's role in the software development process. The linker accepts several types of files as input, including object files, command files, libraries, and partially linked files. The linker creates an executable object module that can be downloaded to one of several development tools or executed by a MSP430 device.

Figure 7-1. The Linker in the MSP430 Software Development Flow



7.3 Invoking the Linker

The general syntax for invoking the linker is:

```
cl430 --run_linker [options] filename1 .... filenamen
```

| | |
|---|---|
| cl430 --run_linker | is the command that invokes the linker. The <code>--run_linker</code> option's short form is <code>-Z</code> . |
| <i>options</i> | can appear anywhere on the command line or in a link command file. (Options are discussed in Section 7.4 .) |
| <i>filename</i> ₁ , <i>filename</i> _n | can be object files, link command files, or archive libraries. The default extension for all input files is <code>.obj</code> ; any other extension must be explicitly specified. The linker can determine whether the input file is an object or ASCII file that contains linker commands. The default output filename is <code>a.out</code> , unless you use the <code>--output_file</code> option to name the output file. |

There are two methods for invoking the linker:

- Specify options and filenames on the command line. This example links two files, `file1.obj` and `file2.obj`, and creates an output module named `link.out`.

```
cl430 --run_linker file1.obj file2.obj --output_file=link.out
```

- Put filenames and options in a link command file. Filenames that are specified inside a link command file must begin with a letter. For example, assume the file `linker.cmd` contains the following lines:

```
--output_file=link.out
file1.obj
file2.obj
```

Now you can invoke the linker from the command line; specify the command filename as an input file:

```
cl430 --run_linker linker.cmd
```

When you use a command file, you can also specify other options and files on the command line. For example, you could enter:

```
cl430 --run_linker --map_file=link.map linker.cmd file3.obj
```

The linker reads and processes a command file as soon as it encounters the filename on the command line, so it links the files in this order: `file1.obj`, `file2.obj`, and `file3.obj`. This example creates an output file called `link.out` and a map file called `link.map`.

For information on invoking the linker for C/C++ files, see [Section 7.17](#).

7.4 Linker Options

Linker options control linking operations. They can be placed on the command line or in a command file. Linker options must be preceded by a hyphen (-). Options can be separated from arguments (if they have them) by an optional space. [Table 7-1](#) summarizes the linker options.

Table 7-1. Linker Options Summary

| Option | Alias | Description | Section |
|----------------------------|-----------|--|---------------------------------|
| --absolute_exe | -a | Produces an absolute, executable module. This is the default; if neither --absolute_exe nor --relocatable is specified, the linker acts as if --absolute_exe were specified. | Section 7.4.2.1 |
| -ar | | Produces a relocatable, executable object module | Section 7.4.2.3 |
| --arg_size | --args | Allocates memory to be used by the loader to pass arguments | Section 7.4.3 |
| --compress_dwarf | | Aggressively reduces the size of DWARF information from input object files | Section 7.4.4 |
| --define | | Predefines <i>name</i> as a preprocessor macro. | Section 7.4.8 |
| --diag_error | | Categorizes the diagnostic identified by <i>num</i> as an error | Section 7.4.5 |
| --diag_remark | | Categorizes the diagnostic identified by <i>num</i> as a remark | Section 7.4.5 |
| --diag_suppress | | Suppresses the diagnostic identified by <i>num</i> | Section 7.4.5 |
| --diag_warning | | Categorizes the diagnostic identified by <i>num</i> as a warning | Section 7.4.5 |
| --disable_auto_rts | | Disables the automatic selection of a run-time-support library | Section 7.4.6 |
| --disable_clink | -j | Disables conditional linking of COFF object modules | Section 7.4.7 |
| --disable_pp | | Disables preprocessing for command files | Section 7.4.8 |
| --display_error_number | | Displays a diagnostic's identifiers along with its text | Section 7.4.5 |
| --entry_point | -e | Defines a global symbol that specifies the primary entry point for the output module | Section 7.4.9 |
| --fill_value | -f | Sets default fill values for holes within output sections; <i>fill_value</i> is a 32-bit constant | Section 7.4.10 |
| --gen_func_subsections | | Puts each function in a separate subsection in the object file | Section 7.4.12 |
| --generate_dead_funcs_list | | Writes a list of the dead functions that were removed by the linker to file <i>fname</i> . | Section 7.4.11 |
| --globalize | | Changes the symbol linkage to global for symbols that match <i>pattern</i> | Section 7.4.16 |
| --heap_size | -heap | Sets heap size (for the dynamic memory allocation in C) to <i>size</i> bytes and defines a global symbol that specifies the heap size. Default = 128 bytes | Section 7.4.13 |
| --hide | | Hides global symbols that match <i>pattern</i> | Section 7.4.14 |
| --issue_remarks | | Issues remarks (nonserious warnings) | Section 7.4.5 |
| --library | -l | Names an archive library or link command <i>filename</i> as linker input | Section 7.4.15 |
| --linker_help | -help | Displays information about syntax and available options | – |
| --localize | | Changes the symbol linkage to local for symbols that match <i>pattern</i> | Section 7.4.16 |
| --make_global | -g | Makes <i>symbol</i> global (overrides -h) | Section 7.4.17 |
| --make_static | -h | Makes all global symbols static | Section 7.4.18 |
| --map_file | -m | Produces a map or listing of the input and output sections, including holes, and places the listing in <i>filename</i> | Section 7.4.19 |
| --mapfile_contents | | Controls the information that appears in the map file. | Section 7.4.20 |
| --no_demangle | | Disables demangling of symbol names in diagnostics | Section 7.4.21 |
| --no_sym_merge | -b | Disables merge of symbolic debugging information in COFF object files | Section 7.4.22 |
| --no_sym_table | -s | Strips symbol table information and line number entries from the output module | Section 7.4.23 |
| --no_warnings | | Suppresses warning diagnostics (errors are still issued) | Section 7.4.5 |
| --output_file | -o | Names the executable output module. The default filename is a.out. | Section 7.4.24 |
| --priority | -priority | Satisfies unresolved references by the first library that contains a definition for that symbol | Section 7.4.26 |
| --ram_model | -cr | Initializes variables at load time | Section 7.4.25 |
| --relocatable | -r | Produces a nonexecutable, relocatable output module | Section 7.4.2.2 |
| --reread_libs | -x | Forces rereading of libraries, which resolves back references | Section 7.4.26 |

Table 7-1. Linker Options Summary (continued)

| Option | Alias | Description | Section |
|---------------------------|-----------|---|----------------------------------|
| --rom_model | -c | Autoinitializes variables at run time | Section 7.4.25 |
| --run_abs | -abs | Produces an absolute listing file | Section 7.4.27 |
| --runtime | | Designates the header include path to use different libraries | Section 7.4.28 |
| --scan_libraries | -scanlibs | Scans all libraries for duplicate symbol definitions | Section 7.4.29 |
| --search_path | -I | Alters library-search algorithms to look in a directory named with <i>pathname</i> before looking in the default location. This option must appear before the --library option. | Section 7.4.15.1 |
| --set_error_limit | | Sets the error limit to <i>num</i> . The linker abandons linking after this number of errors. (The default is 100.) | Section 7.4.5 |
| --stack_size | -stack | Sets C system stack size to <i>size</i> bytes and defines a global symbol that specifies the stack size. Default = 128 bytes | Section 7.4.30 |
| --strict_compatibility | | Performs more conservative and rigorous compatibility checking of input object files | Section 7.4.31 |
| --symbol_map | | Maps symbol references to a symbol definition of a different name | Section 7.4.32 |
| --undef_sym | -u | Places an unresolved external <i>symbol</i> into the output module's symbol table | Section 7.4.33 |
| --undefine | | Removes the preprocessor macro <i>name</i> . | Section 7.4.8 |
| --unhide | | Reveals (un-hides) global symbols that match <i>pattern</i> | Section 7.4.14 |
| --use_hw_mpy[={16 32 F5}] | | Replaces all references to the default integer/long multiply routine with the version of the multiply routine that uses the hardware multiplier support. | Section 7.4.34 |
| --verbose_diagnostics | | Provides verbose diagnostics that display the original source with line-wrap | Section 7.4.5 |
| --warn_sections | -w | Displays a message when an undefined output section is created | Section 7.4.35 |
| --xml_link_info | | Generates a well-formed XML <i>file</i> containing detailed information about the result of a link | Section 7.4.36 |

7.4.1 Wild Cards in File, Section, and Symbol Patterns

The linker allows file, section, and symbol names to be specified using the asterisk (*) and question mark (?) wild cards. Using * matches any number of characters and using ? matches a single character. Using wild cards can make it easier to handle related objects, provided they follow a suitable naming convention.

For example:

```
mp3*.obj      /* matches anything .obj that begins with mp3      */
task?.o*     /* matches task1.obj, task2.obj, taskX.o55, etc. */

SECTIONS
{
    .fast_code: { *.obj(*fast*) } > FAST_MEM
    .vectors   : { vectors.obj(.vector:part1:*) > 0xFFFFFFFF0
    .str_code  : { rts*.lib<str*.obj>(.text) } > S1ROM
}
```

7.4.2 Relocation Capabilities (--absolute_exe and --relocatable Options)

The linker performs relocation, which is the process of adjusting all references to a symbol when the symbol's address changes. The linker supports two options (--absolute_exe and --relocatable) that allow you to produce an absolute or a relocatable output module. The linker also supports a third option (-ar) that allows you to produce an executable, relocatable output module.

When the linker encounters a file that contains no relocation or symbol table information, it issues a warning message (but continues executing). Relinking an absolute file can be successful only if each input file contains no information that needs to be relocated (that is, each file has no unresolved references and is bound to the same virtual address that it was bound to when the linker created it).

7.4.2.1 Producing an Absolute Output Module (--absolute_exe option)

When you use the `--absolute_exe` option without the `--relocatable` option, the linker produces an *absolute, executable* output module. Absolute files contain *no* relocation information. Executable files contain the following:

- Special symbols defined by the linker (see [Section 7.13.4](#))
- An optional header that describes information such as the program entry point
- *No* unresolved references

The following example links `file1.obj` and `file2.obj` and creates an absolute output module called `a.out`:

```
cl430 --run_linker --absolute_exe file1.obj file2.obj
```

The --absolute_exe and --relocatable Options

Note: If you do not use the `--absolute_exe` or the `--relocatable` option, the linker acts as if you specified `--absolute_exe`.

7.4.2.2 Producing a Relocatable Output Module (--relocatable option)

When you use the `-ar` option, the linker retains relocation entries in the output module. If the output module is relocated (at load time) or relinked (by another linker execution), use `--relocatable` to retain the relocation entries.

The linker produces a file that is not executable when you use the `--relocatable` option without the `--absolute_exe` option. A file that is not executable does not contain special linker symbols or an optional header. The file can contain unresolved references, but these references do not prevent creation of an output module.

This example links `file1.obj` and `file2.obj` and creates a relocatable output module called `a.out`:

```
cl430 --run_linker --relocatable file1.obj file2.obj
```

The output file `a.out` can be relinked with other object files or relocated at load time. (Linking a file that will be relinked with other files is called partial linking. For more information, see [Section 7.16](#).)

7.4.2.3 Producing an Executable, Relocatable Output Module (-ar Option)

If you invoke the linker with both the `--absolute_exe` and `--relocatable` options, the linker produces an *executable, relocatable* object module. The output file contains the special linker symbols, an optional header, and all resolved symbol references; however, the relocation information is retained.

This example links `file1.obj` and `file2.obj` and creates an executable, relocatable output module called `xr.out`:

```
cl430 --run_linker -ar file1.obj file2.obj --output_file=xr.out
```

7.4.3 Allocate Memory for Use by the Loader to Pass Arguments (`--arg_size` Option)

The `--arg_size` option instructs the linker to allocate memory to be used by the loader to pass arguments from the command line of the loader to the program. The syntax of the `--arg_size` option is:

`--arg_size=` *size*

The *size* is a number representing the number of bytes to be allocated in target memory for command-line arguments.

By default, the linker creates the `__c_args__` symbol and sets it to -1. When you specify `--arg_size=size`, the following occur:

- The linker creates an uninitialized section named `.args` of *size* bytes.
- The `__c_args__` symbol contains the address of the `.args` section.

The loader and the target boot code use the `.args` section and the `__c_args__` symbol to determine whether and how to pass arguments from the host to the target program. See the *MSP430 Optimizing C/C++ Compiler User's Guide* for information about the loader.

7.4.4 Compress DWARF Information (`--compress_dwarf` Option)

The `--compress_dwarf` option aggressively reduces the size of DWARF information by eliminating duplicate information from input object files. This is the default behavior for COFF object files, and can be disabled for COFF with the legacy `--no_sym_merge` option.

7.4.5 Control Linker Diagnostics

The linker uses certain C/C++ compiler options to control linker-generated diagnostics. The diagnostic options must be specified before the `--run_linker` option.

- | | |
|--|--|
| <code>--diag_error=num</code> | Categorizes the diagnostic identified by <i>num</i> as an error. To determine the numeric identifier of a diagnostic message, use the <code>--display_error_number</code> option first in a separate link. Then use <code>--diag_error=num</code> to recategorize the diagnostic as an error. You can only alter the severity of discretionary diagnostics. |
| <code>--diag_remark=num</code> | Categorizes the diagnostic identified by <i>num</i> as a remark. To determine the numeric identifier of a diagnostic message, use the <code>--display_error_number</code> option first in a separate link. Then use <code>--diag_remark=num</code> to recategorize the diagnostic as a remark. You can only alter the severity of discretionary diagnostics. |
| <code>--diag_suppress=num</code> | Suppresses the diagnostic identified by <i>num</i> . To determine the numeric identifier of a diagnostic message, use the <code>--display_error_number</code> option first in a separate link. Then use <code>--diag_suppress=num</code> to suppress the diagnostic. You can only suppress discretionary diagnostics. |
| <code>--diag_warning=num</code> | Categorizes the diagnostic identified by <i>num</i> as a warning. To determine the numeric identifier of a diagnostic message, use the <code>--display_error_number</code> option first in a separate link. Then use <code>--diag_warning=num</code> to recategorize the diagnostic as a warning. You can only alter the severity of discretionary diagnostics. |
| <code>--display_error_number</code> | Displays a diagnostic's numeric identifier along with its text. Use this option in determining which arguments you need to supply to the diagnostic suppression options (<code>--diag_suppress</code> , <code>--diag_error</code> , <code>--diag_remark</code> , and <code>--diag_warning</code>). This option also indicates whether a diagnostic is discretionary. A discretionary diagnostic is one whose severity can be overridden. A discretionary diagnostic includes the suffix <code>-D</code> ; otherwise, no suffix is present. See the <i>MSP430 Optimizing C/C++ Compiler User's Guide</i> for more information on understanding diagnostic messages. |

| | |
|------------------------------|---|
| --issue_remarks | Issues remarks (nonserious warnings), which are suppressed by default. |
| --no_warnings | Suppresses warning diagnostics (errors are still issued). |
| --set_error_limit=num | Sets the error limit to <i>num</i> , which can be any decimal value. The linker abandons linking after this number of errors. (The default is 100.) |
| --verbose_diagnostics | Provides verbose diagnostics that display the original source with line-wrap and indicate the position of the error in the source line |

7.4.6 Disable Automatic Library Selection (**--disable_auto_rts Option**)

The `--disable_auto_rts` option disables the automatic selection of a run-time-support library. See the *MSP430 Optimizing C/C++ Compiler User's Guide* for details on the automatic selection process.

7.4.7 Disable Conditional Linking (**--disable_clink Option**)

The `--disable_clink` option disables removal of unreferenced sections in COFF object modules. Only sections marked as candidates for removal with the `.clink` assembler directive are affected by conditional linking. See [Conditionally Leave Section Out of Object Module Output](#) for details on setting up conditional linking using the `.clink` directive.

7.4.8 Link Command File Preprocessing (**--disable_pp, --define and --undefine Options**)

The linker preprocesses link command files using a standard C preprocessor. Therefore, the command files can contain well-known preprocessing directives such as `#define`, `#include`, and `#if / #endif`.

Three linker options control the preprocessor:

| | |
|----------------------------|--|
| --disable_pp | Disables preprocessing for command files |
| --define=name[=val] | Predefines <i>name</i> as a preprocessor macro |
| --undefine=name | Removes the macro <i>name</i> |

The compiler has `--define` and `--undefine` options with the same meanings. However, the linker options are distinct; only `--define` and `--undefine` options specified after `--run_linker` are passed to the linker. For example:

```
cl430 --define=FOO=1 main.c --run_linker --define=BAR=2 lnk.cmd
```

The linker sees only the `--define` for `BAR`; the compiler only sees the `--define` for `FOO`.

When one command file `#includes` another, preprocessing context is carried from parent to child in the usual way (that is, macros defined in the parent are visible in the child). However, when a command file is invoked other than through `#include`, either on the command line or by the typical way of being named in another command file, preprocessing context is **not** carried into the nested file. The exception to this is `--define` and `--undefine` options, which apply globally from the point they are encountered. For example:

```
--define GLOBAL
#define LOCAL

#include "incfile.cmd" /* sees GLOBAL and LOCAL */
nestfile.cmd          /* only sees GLOBAL */
```

Two cautions apply to the use of `--define` and `--undefine` in command files. First, they have global effect as mentioned above. Second, since they are not actually preprocessing directives themselves, they are subject to macro substitution, probably with unintended consequences. This effect can be defeated by quoting the symbol name. For example:

```
--define MYSYM=123
--undefine MYSYM /* expands to --undefine 123 (!) */
--undefine "MYSYM" /* ahh, that's better */
```

The linker uses the same search paths to find #include files as it does to find libraries. That is, #include files are searched in the following places:

1. If the #include file name is in quotes (rather than <brackets>), in the directory of the current file
2. In the list of directories specified with --library options or environment variables (see [Section 7.4.15](#))

There are two exceptions: relative pathnames (such as "../name") always search the current directory; and absolute pathnames (such as "/usr/tools/name") bypass search paths entirely.

The linker has the standard built-in definitions for the macros `__FILE__`, `__DATE__`, and `__TIME__`. It does not, however, have the compiler-specific options for the target (`__MSP430__`), version (`__TI_COMPILER_VERSION__`), run-time model, and so on.

7.4.9 Define an Entry Point (`--entry_point` Option)

The memory address at which a program begins executing is called the *entry point*. When a loader loads a program into target memory, the program counter (PC) must be initialized to the entry point; the PC then points to the beginning of the program.

The linker can assign one of four values to the entry point. These values are listed below in the order in which the linker tries to use them. If you use one of the first three values, it must be an external symbol in the symbol table.

- The value specified by the `--entry_point` option. The syntax is:
`--entry_point= global_symbol`
 where *global_symbol* defines the entry point and must be defined as an external symbol of the input files.
- The value of symbol `_c_int00` (if present). The `_c_int00` symbol *must* be the entry point if you are linking code produced by the C compiler.
- The value of symbol `_main` (if present)
- 0 (default value)

This example links file1.obj and file2.obj. The symbol `begin` is the entry point; `begin` must be defined as external in file1 or file2.

```
c1430 --run_linker --entry_point=begin file1.obj file2.obj
```

7.4.10 Set Default Fill Value (`--fill_value` Option)

The `--fill_value` option fills the holes formed within output sections. The syntax for the `--fill_value` option is:

```
--fill_value=value
```

The argument *value* is a 32-bit constant (up to eight hexadecimal digits). If you do not use `--fill_value`, the linker uses 0 as the default fill value.

This example fills holes with the hexadecimal value ABCDABCD:

```
c1430 --run_linker --fill_value=0xABCDABCD file1.obj file2.obj
```

7.4.11 Generate List of Dead Functions (`--generate_dead_funcs_list` Option)

The `--generate_dead_funcs_list` option creates a list of functions that are never referenced (dead) and writes the list to the specified file. If no filename is specified, the default filename `dead_funcs.txt` is used. The syntax for the option is:

```
--generate_dead_funcs_list=filename
```

Refer to the *MSP430 Optimizing C/C++ Compiler User's Guide* for details on the `--generate_dead_funcs_list` option.

7.4.12 Using Function Subsections (`--gen_func_subsections` Option)

When the linker places code into an executable file, it allocates all the functions in a single source file as a group. This means that if any function in a file needs to be linked into an executable, then all the functions in the file are linked in. This can be undesirable if a file contains many functions and only a few are required for an executable.

This situation may exist in libraries where a single file contains multiple functions, but the application only needs a subset of those functions. An example is a library .obj file that contains a signed divide routine and an unsigned divide routine. If the application requires only signed division, then only the signed divide routine is required for linking. By default, both the signed and unsigned routines are linked in since they exist in the same .obj file.

The `--gen_func_subsections` compiler option remedies this problem by placing each function in a file in its own subsection. Thus, only the functions that are referenced in the application are linked into the final executable. This can result in an overall code size reduction.

7.4.13 Define Heap Size (`--heap_size` Option)

The C/C++ compiler uses an uninitialized section called .system for the C run-time memory pool used by `malloc()`. You can set the size of this memory pool at link time by using the `--heap_size` option. The syntax for the `--heap_size` option is:

`--heap_size= size`

The *size* must be a constant. This example defines a 4K byte heap:

```
cl430 --run_linker --heap_size=0x1000 /* defines a 4k heap (.system section)*/
```

The linker creates the .system section only if there is a .system section in an input file.

The linker also creates a global symbol `__SYSTEM_SIZE` and assigns it a value equal to the size of the heap. The default size is 128 bytes.

For more information about C/C++ linking, see [Section 7.17](#).

7.4.14 Hiding Symbols

Symbol hiding prevents the symbol from being listed in the output file's symbol table. The linker supports symbol hiding through the `--hide` and `--unhide` options.

The syntax for these options are:

`--hide=' pattern '`

`--unhide=' pattern '`

The *pattern* is a string with optional wildcards `?` or `*`. Use `?` to match a single character and use `*` to match zero or more characters.

The `--hide` option hides global symbols that match the *pattern*. It hides the symbols matching the pattern by changing the name to an empty string.

The `--unhide` option reveals (un-hides) global symbols that match the *pattern* that are hidden by the `--hide` option. The `--unhide` option excludes symbols that match pattern from symbol hiding provided the pattern defined by `--unhide` is more restrictive than the pattern defined by `--hide`.

These options have the following properties:

- The `--hide` and `--unhide` options can be specified more than once on the command line.
- The order of `--hide` and `--unhide` has no significance.
- A symbol is matched by only one pattern defined by either `--hide` or `--unhide`.
- A symbol is matched by the most restrictive pattern. Pattern A is considered more restrictive than Pattern B, if Pattern A matches a narrower set than Pattern B.
- It is an error if a symbol matches patterns from `--hide` and `--unhide` and if one does not supersede other. Pattern A supersedes pattern B if A can match everything B can, and some more. If Pattern A supersedes Pattern B, then Pattern B is said to more restrictive than Pattern A.

- These options affect final and partial linking.

In map files these symbols are listed under the Hidden Symbols heading.

7.4.15 Alter the Library Search Algorithm (`--library` Option, `--search_path` Option, and `MSP430_C_DIR` Environment Variable)

Usually, when you want to specify a file as linker input, you simply enter the filename; the linker looks for the file in the current directory. For example, suppose the current directory contains the library object.lib. Assume that this library defines symbols that are referenced in the file file1.obj. This is how you link the files:

```
cl430 --run_linker file1.obj object.lib
```

If you want to use a file that is not in the current directory, use the `--library` linker option. The `--library` option's short form is `-l`. The syntax for this option is:

```
--library=[pathname] filename
```

The *filename* is the name of an archive, an object file, or link command file. You can specify up to 128 search paths.

The `--library` option is not required when one or more members of an object library are specified for input to an output section. For more information about allocating archive members, see [Section 7.8.7](#).

You can augment the linker's directory search algorithm by using the `--search_path` linker option or the `MSP430_C_DIR` environment variable. The linker searches for object libraries and command files in the following order:

1. It searches directories named with the `--search_path` linker option. The `--search_path` option must appear before the `--library` option on the command line or in a command file.
2. It searches directories named with `MSP430_C_DIR`.
3. If `MSP430_C_DIR` is not set, it searches directories named with the assembler's `MSP430_A_DIR` environment variable.
4. It searches the current directory.

7.4.15.1 Name an Alternate Library Directory (`--search_path` Option)

The `--search_path` option names an alternate directory that contains input files. The `--search_path` option's short form is `-I`. The syntax for this option is:

```
--search_path=pathname
```

The *pathname* names a directory that contains input files.

When the linker is searching for input files named with the `--library` option, it searches through directories named with `--search_path` first. Each `--search_path` option specifies only one directory, but you can have several `--search_path` options per invocation. When you use the `--search_path` option to name an alternate directory, it must precede any `--library` option used on the command line or in a command file.

For example, assume that there are two archive libraries called `r.lib` and `lib2.lib` that reside in `ld` and `ld2` directories. The table below shows the directories that `r.lib` and `lib2.lib` reside in, how to set environment variable, and how to use both libraries during a link. Select the row for your operating system:

| Operating System | Enter |
|---------------------|---|
| UNIX (Bourne shell) | <code>cl430 --run_linker f1.obj f2.obj --search_path=/ld --search_path=/ld2 --library=r.lib --library=lib2.lib</code> |
| Windows | <code>cl430 --run_linker f1.obj f2.obj --search_path=\ld --search_path=\ld2 --library=r.lib --library=lib2.lib</code> |

7.4.15.2 Name an Alternate Library Directory (MSP430_C_DIR Environment Variable)

An environment variable is a system symbol that you define and assign a string to. The linker uses an environment variable named MSP430_C_DIR to name alternate directories that contain object libraries. The command syntaxes for assigning the environment variable are:

| Operating System | Enter |
|---------------------|---|
| UNIX (Bourne shell) | MSP430_C_DIR=" <i>pathname₁</i> ; <i>pathname₂</i> ; ... "; export MSP430_C_DIR |
| Windows | set MSP430_C_DIR= <i>pathname₁</i> ; <i>pathname₂</i> ; ... |

The *pathnames* are directories that contain input files. Use the --library linker option on the command line or in a command file to tell the linker which library or link command file to search for. The pathnames must follow these constraints:

- Pathnames must be separated with a semicolon.
- Spaces or tabs at the beginning or end of a path are ignored. For example the space before and after the semicolon in the following is ignored:

```
set MSP430_C_DIR= c:\path\one\to\tools ; c:\path\two\to\tools
```

- Spaces and tabs are allowed within paths to accommodate Windows directories that contain spaces. For example, the pathnames in the following are valid:

```
set MSP430_C_DIR=c:\first path\to\tools;d:\second path\to\tools
```

In the example below, assume that two archive libraries called r.lib and lib2.lib reside in ld and ld2 directories. The table below shows how to set the environment variable, and how to use both libraries during a link. Select the row for your operating system:

| Operating System | Invocation Command |
|---------------------|--|
| UNIX (Bourne shell) | MSP430_C_DIR="/ld ;/ld2"; export MSP430_C_DIR; cl430 --run_linker f1.obj f2.obj --library=r.lib --library=lib2.lib |
| Windows | MSP430_C_DIR=ld;ld2 cl430 --run linker f1.obj f2.obj --library=r.lib --library=lib2.lib |

The environment variable remains set until you reboot the system or reset the variable by entering:

| Operating System | Enter |
|---------------------|--------------------|
| UNIX (Bourne shell) | unset MSP430_C_DIR |
| Windows | set MSP430_C_DIR= |

The assembler uses an environment variable named MSP430_A_DIR to name alternate directories that contain copy/include files or macro libraries. If MSP430_C_DIR is not set, the linker searches for object libraries in the directories named with MSP430_A_DIR. For information about MSP430_A_DIR, see [Section 3.4.2](#). For more information about object libraries, see [Section 7.6](#).

7.4.16 Change Symbol Localization

Symbol localization changes symbol linkage from global to local (static). The linker supports symbol localization through the `--localize` and `--globalize` linker options.

The syntax for these options are:

`--localize=' pattern '`

`--globalize=' pattern '`

The *pattern* is a string with optional wild cards `?` or `*`. Use `?` to match a single character and use `*` to match zero or more characters.

The `--localize` option changes the symbol linkage to local for symbols matching the *pattern*.

The `--globalize` option changes the symbol linkage to global for symbols matching the *pattern*. The `--globalize` option only affects symbols that are localized by the `--localize` option. The `--globalize` option excludes symbols that match the pattern from symbol localization, provided the pattern defined by `--globalize` is more restrictive than the pattern defined by `--localize`.

These options have the following properties:

- The `--localize` and `--globalize` options can be specified more than once on the command line.
- The order of `--localize` and `--globalize` options has no significance.
- A symbol is matched by only one pattern defined by either `--localize` or `--globalize`.
- A symbol is matched by the most restrictive pattern. Pattern A is considered more restrictive than Pattern B, if Pattern A matches a narrower set than Pattern B.
- It is an error if a symbol matches patterns from `--localize` and `--globalize` and if one does not supersede other. Pattern A supersedes pattern B if A can match everything B can, and some more. If Pattern A supersedes Pattern B, then Pattern B is said to more restrictive than Pattern A.
- These options affect final and partial linking.

In map files these symbols are listed under the Localized Symbols heading.

7.4.17 Make a Symbol Global (`--make_global` Option)

The `--make_static` option makes all global symbols static. If you have a symbol that you want to remain global and you use the `--make_static` option, you can use the `--make_global` option to declare that symbol to be global. The `--make_global` option overrides the effect of the `--make_static` option for the symbol that you specify. The syntax for the `--make_global` option is:

`--make_global= global_symbol`

7.4.18 Make All Global Symbols Static (`--make_static` Option)

The `--make_static` option makes all global symbols static. Static symbols are not visible to externally linked modules. By making global symbols static, global symbols are essentially hidden. This allows external symbols with the same name (in different files) to be treated as unique.

The `--make_static` option effectively nullifies all `.global` assembler directives. All symbols become local to the module in which they are defined, so no external references are possible. For example, assume `file1.obj` and `file2.obj` both define global symbols called `EXT`. By using the `--make_static` option, you can link these files without conflict. The symbol `EXT` defined in `file1.obj` is treated separately from the symbol `EXT` defined in `file2.obj`.

```
cl430 --run_linker --make_static file1.obj file2.obj
```

7.4.19 Create a Map File (`--map_file` Option)

The `--map_file` option creates a linker map listing and puts it in *filename*. The syntax for the `--map_file` option is:

`--map_file= filename`

The linker map describes:

- Memory configuration
- Input and output section allocation
- Linker-generated copy tables
- The addresses of external symbols after they have been relocated
- Hidden and localized symbols

The map file contains the name of the output module and the entry point; it can also contain up to three tables:

- A table showing the new memory configuration if any nondefault memory is specified (memory configuration). The table has the following columns; this information is generated on the basis of the information in the MEMORY directive in the link command file:
 - **Name.** This is the name of the memory range specified with the MEMORY directive.
 - **Origin.** This specifies the starting address of a memory range.
 - **Length.** This specifies the length of a memory range.
 - **Unused.** This specifies the total amount of unused (available) memory in that memory area.
 - **Attributes.** This specifies one to four attributes associated with the named range:
 - R specifies that the memory can be read.
 - W specifies that the memory can be written to.
 - X specifies that the memory can contain executable code.
 - I specifies that the memory can be initialized.

For more information about the MEMORY directive, see [Section 7.7](#).

- A table showing the linked addresses of each output section and the input sections that make up the output sections (section allocation map). This table has the following columns; this information is generated on the basis of the information in the SECTIONS directive in the link command file:
 - **Output section.** This is the name of the output section specified with the SECTIONS directive.
 - **Origin.** The first origin listed for each output section is the starting address of that output section. The indented origin value is the starting address of that portion of the output section.
 - **Length.** The first length listed for each output section is the length of that output section. The indented length value is the length of that portion of the output section.
 - **Attributes/input sections.** This lists the input file or value associated with an output section. If the input section could not be allocated, the map file will indicate this with "FAILED TO ALLOCATE".

For more information about the SECTIONS directive, see [Section 7.8](#).

- A table showing each external symbol and its address sorted by symbol name.
- A table showing each external symbol and its address sorted by symbol address.

This following example links file1.obj and file2.obj and creates a map file called map.out:

```
c1430 --run_linker file1.obj file2.obj --map_file=map.out
```

[Example 7-25](#) shows an example of a map file.

7.4.20 Managing Map File Contents (--mapfile_contents Option)

The --mapfile_contents option assists with managing the content of linker-generated map files. The syntax for the --mapfile_contents option is:

```
--mapfile_contents=filter[, filter]
```

When the --map_file option is specified, the linker produces a map file containing information about memory usage, placement information about sections that were created during a link, details about linker-generated copy tables, and symbol values.

The new --mapfile_contents option provides a mechanism for you to control what information is included in or excluded from a map file. When you specify --mapfile_contents=help from the command line, a help screen listing available filter options is displayed.

The following filter options are available:

| Attribute | Description | Default State |
|-------------|-------------------------------|---------------|
| copytables | Copy tables | On |
| entry | Entry point | On |
| load_addr | Display load addresses | Off |
| memory | Memory ranges | On |
| sections | Sections | On |
| sym_defs | Defined symbols per file | Off |
| sym_name | Symbols sorted by name | On |
| sym_runaddr | Symbols sorted by run address | On |
| all | Enables all attributes | |
| none | Disables all attributes | |

The --mapfile_contents option controls display filter settings by specifying a comma-delimited list of display attributes. When prefixed with the word no, an attribute is disabled instead of enabled. For example:

```
--mapfile_contents=copytables,noentry
--mapfile_contents=all,nocopytables
--mapfile_contents=none,entry
```

By default, those sections that are currently included in the map file when the --map_file option is specified are included. The filters specified in the --mapfile_contents options are processed in the order that they appear in the command line. In the third example above, the first filter, none, clears all map file content. The second filter, entry, then enables information about entry points to be included in the generated map file. That is, when --mapfile_contents=none,entry is specified, the map file contains *only* information about entry points.

There are two new filters included with the --mapfile_contents option, load_addr and sym_defs. These are both disabled by default. If you turn on the load_addr filter, the map file includes the load address of symbols that are included in the symbol list in addition to the run address (if the load address is different from the run address).

The sym_defs filter can be used to include information about all static and global symbols defined in an application on a file by file basis. You may find it useful to replace the sym_name and sym_runaddr sections of the map file with the sym_defs section by specifying the following --mapfile_contents option:

```
--mapfile_contents=nosym_name,nosym_runaddr,sym_defs
```

7.4.21 Disable Name Demangling (`--no_demangle`)

By default, the linker uses demangled symbol names in diagnostics. For example:

```
undefined symbol          first referenced in file
ANewClass::getValue()    test.obj
```

The `--no_demangle` option disables the demangling of symbol names in diagnostics. For example:

```
undefined symbol          first referenced in file
_ZN9ANewClass8getValueEv  test.obj
```

7.4.22 Disable Merge of Symbolic Debugging Information (`--no_sym_merge Option`)

By default, the linker eliminates duplicate entries of symbolic debugging information. Such duplicate information is commonly generated when a C program is compiled for debugging. For example:

```
-[ header.h ]-
typedef struct
{
    <define some structure members>
} XYZ;

-[ f1.c ]-
#include "header.h"
...

-[ f2.c ]-
#include "header.h"
...
```

When these files are compiled for debugging, both `f1.obj` and `f2.obj` have symbolic debugging entries to describe type `XYZ`. For the final output file, only one set of these entries is necessary. The linker eliminates the duplicate entries automatically.

Use the COFF only `--no_sym_merge` option if you want the linker to keep such duplicate entries in COFF object files. Using the `--no_sym_merge` option has the effect of the linker running faster and using less machine memory.

7.4.23 Strip Symbolic Information (`--no_sym_table Option`)

The `--no_sym_table` option creates a smaller output module by omitting symbol table information and line number entries. The `--no_sym_table` option is useful for production applications when you do not want to disclose symbolic information to the consumer.

This example links `file1.obj` and `file2.obj` and creates an output module, stripped of line numbers and symbol table information, named `nosym.out`:

```
c1430 --run_linker --output_file=nosym.out --no_sym_table file1.obj file2.obj
```

Using the `--no_sym_table` option limits later use of a symbolic debugger.

Stripping Symbolic Information

Note: To remove symbol table information, use the `strip430` utility as described in [Section 10.4](#). The `--no_sym_table` option is deprecated.

7.4.24 Name an Output Module (`--output_file` Option)

The linker creates an output module when no errors are encountered. If you do not specify a filename for the output module, the linker gives it the default name `a.out`. If you want to write the output module to a different file, use the `--output_file` option. The syntax for the `--output_file` option is:

```
--output_file= filename
```

The *filename* is the new output module name.

This example links `file1.obj` and `file2.obj` and creates an output module named `run.out`:

```
c1430 --run_linker --output_file=run.out file1.obj file2.obj
```

7.4.25 C Language Options (`--ram_model` and `--rom_model` Options)

The `--ram_model` and `--rom_model` options cause the linker to use linking conventions that are required by the C compiler.

- The `--ram_model` option tells the linker to initialize variables at load time.
- The `--rom_model` option tells the linker to autoinitialize variables at run time.

For more information, see [Section 7.17](#), [Section 7.17.4](#), and [Section 7.17.5](#).

7.4.26 Exhaustively Read and Search Libraries (`--reread_libs` and `--priority` Options)

There are two ways to exhaustively search for unresolved symbols:

- Reread libraries if you cannot resolve a symbol reference (`--reread_libs`).
- Search libraries in the order that they are specified (`--priority`).

The linker normally reads input files, including archive libraries, only once when they are encountered on the command line or in the command file. When an archive is read, any members that resolve references to undefined symbols are included in the link. If an input file later references a symbol defined in a previously read archive library, the reference is not resolved.

With the `--reread_libs` option, you can force the linker to reread all libraries. The linker rereads libraries until no more references can be resolved. Linking using `--reread_libs` may be slower, so you should use it only as needed. For example, if `a.lib` contains a reference to a symbol defined in `b.lib`, and `b.lib` contains a reference to a symbol defined in `a.lib`, you can resolve the mutual dependencies by listing one of the libraries twice, as in:

```
c1430 --run_linker --library=a.lib --library=b.lib --library=a.lib
```

or you can force the linker to do it for you:

```
c1430 --run_linker --reread_libs --library=a.lib --library=b.lib
```

The `--priority` option provides an alternate search mechanism for libraries. Using `--priority` causes each unresolved reference to be satisfied by the first library that contains a definition for that symbol. For example:

```
objfile  references A
lib1     defines B
lib2     defines A, B; obj defining A references B

% c1430 --run_linker objfile lib1 lib2
```

Under the existing model, `objfile` resolves its reference to `A` in `lib2`, pulling in a reference to `B`, which resolves to the `B` in `lib2`.

Under `--priority`, `objfile` resolves its reference to `A` in `lib2`, pulling in a reference to `B`, but now `B` is resolved by searching the libraries in order and resolves `B` to the first definition it finds, namely the one in `lib1`.

The `--priority` option is useful for libraries that provide overriding definitions for related sets of functions in other libraries without having to provide a complete version of the whole library.

For example, suppose you want to override versions of `malloc` and `free` defined in the `rts430.lib` without providing a full replacement for `rts430.lib`. Using `--priority` and linking your new library before `rts430.lib` guarantees that all references to `malloc` and `free` resolve to the new library.

The `--priority` option is intended to support linking programs with DSP/BIOS where situations like the one illustrated above occur.

7.4.27 Create an Absolute Listing File (`--run_abs` Option)

The `--run_abs` option produces an output file for each file that was linked. These files are named with the input filenames and an extension of `.abs`. Header files, however, do not generate a corresponding `.abs` file.

7.4.28 Designate Header Path (`--runtime` Option)

The `--runtime` option designates the header include path to use different libraries. The syntax for the `--runtime` option is:

`--runtime=pathname`

7.4.29 Scan All Libraries for Duplicate Symbol Definitions (`--scan_libraries`)

The `--scan_libraries` option scans all libraries during a link looking for duplicate symbol definitions to those symbols that are actually included in the link. The scan does not consider absolute symbols or symbols defined in COMDAT sections. The `--scan_libraries` option helps determine those symbols that were actually chosen by the linker over other existing definitions of the same symbol in a library.

The library scanning feature can be used to check against unintended resolution of a symbol reference to a definition when multiple definitions are available in the libraries.

7.4.30 Define Stack Size (`--stack_size` Option)

The MSP430 C/C++ compiler uses an uninitialized section, `.stack`, to allocate space for the run-time stack. You can set the size of this section in bytes at link time with the `--stack_size` option. The syntax for the `--stack_size` option is:

`--stack_size= size`

The *size* must be a constant and is in bytes. This example defines a 4K byte stack:

```
c1430 --run_linker --stack_size=0x1000 /* defines a 4K heap (.stack section)*/
```

If you specified a different stack size in an input section, the input section stack size is ignored. Any symbols defined in the input section remain valid; only the stack size is different.

When the linker defines the `.stack` section, it also defines a global symbol, `__STACK_SIZE`, and assigns it a value equal to the size of the section. The default software stack size is 128 bytes.

7.4.31 Enforce Strict Compatibility (`--strict_compatibility` Option)

The linker performs more conservative and rigorous compatibility checking of input object files when you specify the `--strict_compatibility` option. Using this option guards against additional potential compatibility issues, but may signal false compatibility errors when linking in object files built with an older toolset, or with object files built with another compiler vendor's toolset. To avoid issues with legacy libraries, the `--strict_compatibility` option is turned off by default.

7.4.32 Mapping of Symbols (--symbol_map Option)

Symbol mapping allows a symbol reference to be resolved by a symbol with a different name. Symbol mapping allows functions to be overridden with alternate definitions. This feature can be used to patch in alternate implementations, which provide patches (bug fixes) or alternate functionality. The syntax for the --symbol_map option is:

```
--symbol_map=refname=defname
```

For example, the following code makes the linker resolve any references to foo by the definition foo_patch:

```
--symbol_map='foo=foo_patch'
```

7.4.33 Introduce an Unresolved Symbol (--undef_sym Option)

The --undef_sym option introduces an unresolved symbol into the linker's symbol table. This forces the linker to search a library and include the member that defines the symbol. The linker must encounter the --undef_sym option *before* it links in the member that defines the symbol. The syntax for the --undef_sym option is:

```
--undef_sym= symbol
```

For example, suppose a library named rts430.lib contains a member that defines the symbol symtab; none of the object files being linked reference symtab. However, suppose you plan to relink the output module and you want to include the library member that defines symtab in this link. Using the --undef_sym option as shown below forces the linker to search rts430.lib for the member that defines symtab and to link in the member.

```
cl430 --run_linker --undef_sym=symtab file1.obj file2.obj rts430.lib
```

If you do not use --undef_sym, this member is not included, because there is no explicit reference to it in file1.obj or file2.obj.

7.4.34 Replace Multiply Routine With Hardware Multiplier Routine (--use_hw_mpy)

There are two versions of each multiply routine included in the run-time library. By default the compiler generates references to the version that does not use the hardware multiplier peripheral that is available on some versions of the MSP430 device. When compiling for a device where the hardware multiplier is available, use the --use_hw_mpy linker option. The syntax for the --use_hw_mpy option is:

```
--use_hw_mpy[={16|32|F5}]
```

The --use_hw_mpy option causes the linker to replace all references to the default multiply routine with the version of the multiply routine that uses the hardware multiplier support. The optional argument indicates which version of the hardware multiply is being used and must be one of the following:

| | |
|----|--------------------------------------|
| 16 | 16-bit hardware multiplier (default) |
| 32 | F4xxx 32-bit hardware multiplier |
| F5 | F5xxx 32-bit hardware multiplier |

For more information regarding the hardware multiplier, see the *MSP430x1xx Family User's Guide*, the *MSP430x3xx Family User's Guide*, the *MSP430x4xx Family User's Guide*, and the *MSP430x5xx Family User's Guide*.

7.4.35 Display a Message When an Undefined Output Section Is Created (--warn_sections Option)

In a link command file, you can set up a SECTIONS directive that describes how input sections are combined into output sections. However, if the linker encounters one or more input sections that do not have a corresponding output section defined in the SECTIONS directive, the linker combines the input sections that have the same name into an output section with that name. By default, the linker does not display a message to tell you that this occurred.

You can use the --warn_sections option to cause the linker to display a message when it creates a new output section.

For more information about the SECTIONS directive, see [Section 7.8](#). For more information about the default actions of the linker, see [Section 7.12](#).

7.4.36 Generate XML Link Information File (--xml_link_info Option)

The linker supports the generation of an XML link information file through the --xml_link_info=*file* option. This option causes the linker to generate a well-formed XML file containing detailed information about the result of a link. The information included in this file includes all of the information that is currently produced in a linker generated map file.

See [Appendix B](#) for specifics on the contents of the generated XML file.

7.5 Linker Command Files

Linker command files allow you to put linking information in a file; this is useful when you invoke the linker often with the same information. Linker command files are also useful because they allow you to use the MEMORY and SECTIONS directives to customize your application. You must use these directives in a command file; you cannot use them on the command line.

Linker command files are ASCII files that contain one or more of the following:

- Input filenames, which specify object files, archive libraries, or other command files. (If a command file calls another command file as input, this statement must be the *last* statement in the calling command file. The linker does not return from called command files.)
- Linker options, which can be used in the command file in the same manner that they are used on the command line
- The MEMORY and SECTIONS linker directives. The MEMORY directive defines the target memory configuration (see [Section 7.7](#)). The SECTIONS directive controls how sections are built and allocated (see [Section 7.8](#).)
- Assignment statements, which define and assign values to global symbols

To invoke the linker with a command file, enter the `cl430 --run_linker` command and follow it with the name of the command file:

```
cl430 --run_linker command_filename
```

The linker processes input files in the order that it encounters them. If the linker recognizes a file as an object file, it links the file. Otherwise, it assumes that a file is a command file and begins reading and processing commands from it. Command filenames are case sensitive, regardless of the system used.

[Example 7-1](#) shows a sample link command file called `link.cmd`.

Example 7-1. Linker Command File

```
a.obj          /* First input filename      */
b.obj          /* Second input filename       */
--output_file=prog.out /* Option to specify output file */
--map_file=prog.map /* Option to specify map file   */
```

The sample file in [Example 7-1](#) contains only filenames and options. (You can place comments in a command file by delimiting them with `/*` and `*/`.) To invoke the linker with this command file, enter:

```
cl430 --run_linker link.cmd
```

You can place other parameters on the command line when you use a command file:

```
cl430 --run_linker --relocatable link.cmd c.obj d.obj
```

The linker processes the command file as soon as it encounters the filename, so `a.obj` and `b.obj` are linked into the output module before `c.obj` and `d.obj`.

You can specify multiple command files. If, for example, you have a file called `names.lst` that contains filenames and another file called `dir.cmd` that contains linker directives, you could enter:

```
cl430 --run_linker names.lst dir.cmd
```

One command file can call another command file; this type of nesting is limited to 16 levels. If a command file calls another command file as input, this statement must be the *last* statement in the calling command file.

Blanks and blank lines are insignificant in a command file except as delimiters. This also applies to the format of linker directives in a command file. [Example 7-2](#) shows a sample command file that contains linker directives.

Example 7-2. Command File With Linker Directives

```

a.obj b.obj c.obj          /* Input filenames      */
--output_file=prog.out    /* Options          */
--map_file=prog.map

MEMORY                    /* MEMORY directive  */
{
  RAM:      origin = 0x0200    length = 0x0100
  EXT_MEM:  origin = 0x1000    length = 0x1000
}

SECTIONS                  /* SECTIONS directive */
{
  .text: > EXT_MEM
  .data: > EXT_MEM
  .bss: > RAM
}

```

For more information, see [Section 7.7](#) for the MEMORY directive, and [Section 7.8](#) for the SECTIONS directive.

7.5.1 Reserved Names in Linker Command Files

The following names are reserved as keywords for linker directives. Do not use them as symbol or section names in a command file.

| | | | | |
|-------|-----------------|--------|--------|----------|
| align | DSECT | len | o | RUN |
| ALIGN | f | length | org | SECTIONS |
| attr | fill | LENGTH | origin | spare |
| ATTR | FILL | load | ORIGIN | type |
| block | group | LOAD | range | TYPE |
| BLOCK | GROUP | MEMORY | run | UNION |
| COPY | I (lowercase L) | NOLOAD | | |

7.5.2 Constants in Linker Command Files

You can specify constants with either of two syntax schemes: the scheme used for specifying decimal, octal, or hexadecimal constants used in the assembler (see [Section 3.6](#)) or the scheme used for integer constants in C syntax.

Examples:

| Format | Decimal | Octal | Hexadecimal |
|------------------|---------|-------|-------------|
| Assembler format | 32 | 40q | 020h |
| C format | 32 | 040 | 0x20 |

7.6 Object Libraries

An object library is a partitioned archive file that contains object files as members. Usually, a group of related modules are grouped together into a library. When you specify an object library as linker input, the linker includes any members of the library that define existing unresolved symbol references. You can use the archiver to build and maintain libraries. [Section 6.1](#) contains more information about the archiver.

Using object libraries can reduce link time and the size of the executable module. Normally, if an object file that contains a function is specified at link time, the file is linked whether the function is used or not; however, if that same function is placed in an archive library, the file is included only if the function is referenced.

The order in which libraries are specified is important, because the linker includes only those members that resolve symbols that are undefined at the time the library is searched. The same library can be specified as often as necessary; it is searched each time it is included. Alternatively, you can use the `--reread_libs` option to reread libraries until no more references can be resolved (see [Section 7.4.26](#)). A library has a table that lists all external symbols defined in the library; the linker searches through the table until it determines that it cannot use the library to resolve any more references.

The following examples link several files and libraries, using these assumptions:

- Input files `f1.obj` and `f2.obj` both reference an external function named `clrscr`.
- Input file `f1.obj` references the symbol `origin`.
- Input file `f2.obj` references the symbol `fillclr`.
- Member 0 of library `libc.lib` contains a definition of `origin`.
- Member 3 of library `liba.lib` contains a definition of `fillclr`.
- Member 1 of both libraries defines `clrscr`.

If you enter:

```
c1430 --run_linker f1.obj f2.obj liba.lib libc.lib
```

then:

- Member 1 of `liba.lib` satisfies the `f1.obj` and `f2.obj` references to `clrscr` because the library is searched and the definition of `clrscr` is found.
- Member 0 of `libc.lib` satisfies the reference to `origin`.
- Member 3 of `liba.lib` satisfies the reference to `fillclr`.

If, however, you enter:

```
c1430 --run_linker f1.obj f2.obj libc.lib liba.lib
```

then the references to `clrscr` are satisfied by member 1 of `libc.lib`.

If none of the linked files reference symbols defined in a library, you can use the `--undef_sym` option to force the linker to include a library member. (See [Section 7.4.33](#).) The next example creates an undefined symbol `rot1` in the linker's global symbol table:

```
c1430 --run_linker --undef_sym=rot1 libc.lib
```

If any member of `libc.lib` defines `rot1`, the linker includes that member.

Library members are allocated according to the `SECTIONS` directive default allocation algorithm; see [Section 7.8](#).

[Section 7.4.15](#) describes methods for specifying directories that contain object libraries.

7.7 The MEMORY Directive

The linker determines where output sections are allocated into memory; it must have a model of target memory to accomplish this. The MEMORY directive allows you to specify a model of target memory so that you can define the types of memory your system contains and the address ranges they occupy. The linker maintains the model as it allocates output sections and uses it to determine which memory locations can be used for object code.

The memory configurations of MSP430 systems differ from application to application. The MEMORY directive allows you to specify a variety of configurations. After you use MEMORY to define a memory model, you can use the SECTIONS directive to allocate output sections into defined memory.

For more information, see [Section 2.3](#) and [Section 2.4](#).

7.7.1 Default Memory Model

The assembler inserts a field in the output file's header, identifying the MSP430 device. The linker reads this information from the object file's header. If you do not use the MEMORY directive, the linker uses a default memory model specific to the named device. For more information about the default memory model, see [Section 7.12](#)

7.7.2 MEMORY Directive Syntax

The MEMORY directive identifies ranges of memory that are physically present in the target system and can be used by a program. Each range has several characteristics:

- Name
- Starting address
- Length
- Optional set of attributes
- Optional fill specification

When you use the MEMORY directive, be sure to identify all memory ranges that are available for loading code. Memory defined by the MEMORY directive is configured; any memory that you do not explicitly account for with MEMORY is unconfigured. The linker does not place any part of a program into unconfigured memory. You can represent nonexistent memory spaces by simply not including an address range in a MEMORY directive statement.

The MEMORY directive is specified in a command file by the word MEMORY (uppercase), followed by a list of memory range specifications enclosed in braces. The MEMORY directive in [Example 7-3](#) defines a system that has 0xEEE0 bytes of flash memory at address 0x1100 and 0x0800 bytes of RAM at address 0x0200.

Example 7-3. The MEMORY Directive

```

/*****
/*      Sample command file with MEMORY directive      */
/*****
file1.obj  file2.obj          /*      Input files      */
--output_file=prog.out      /*      Options      */

MEMORY
{
  FLASH (RX):  origin = 0x1100  length = 0xEEE0
  RAM (RX):    origin = 0x0200  length = 0x0800
}

```

The general syntax for the MEMORY directive is:

MEMORY

```
{
    name 1 [(attr)] : origin = constant, length = constant [, fill = constant]
    .
    .
    name n [(attr)] : origin = constant, length = constant [, fill = constant]
}
```

| | |
|---------------|--|
| name | names a memory range. A memory name can be one to 64 characters; valid characters include A-Z, a-z, \$, ., and _. The names have no special significance to the linker; they simply identify memory ranges. Memory range names are internal to the linker and are not retained in the output file or in the symbol table. All memory ranges must have unique names and must not overlap. |
| attr | specifies one to four attributes associated with the named range. Attributes are optional; when used, they must be enclosed in parentheses. Attributes restrict the allocation of output sections into certain memory ranges. If you do not use any attributes, you can allocate any output section into any range with no restrictions. Any memory for which no attributes are specified (including all memory in the default model) has all four attributes. Valid attributes are: <ul style="list-style-type: none"> R specifies that the memory can be read. W specifies that the memory can be written to. X specifies that the memory can contain executable code. I specifies that the memory can be initialized. |
| origin | specifies the starting address of a memory range; enter as <i>origin</i> , <i>org</i> , or <i>o</i> . The value, specified in bytes, is a 16-bit constant and can be decimal, octal, or hexadecimal. |
| length | specifies the length of a memory range; enter as <i>length</i> , <i>len</i> , or <i>l</i> . The value, specified in bytes, is a 32-bit constant and can be decimal, octal, or hexadecimal. |
| fill | specifies a fill character for the memory range; enter as <i>fill</i> or <i>f</i> . Fills are optional. The value is a integer constant and can be decimal, octal, or hexadecimal. The fill value is used to fill areas of the memory range that are not allocated to a section. |

Filling Memory Ranges

Note: If you specify fill values for large memory ranges, your output file will be very large because filling a memory range (even with 0s) causes raw data to be generated for all unallocated blocks of memory in the range.

The following example specifies a memory range with the R and W attributes and a fill constant of 0FFFFh:

```
MEMORY
{
    RFILE (RW) : o = 0x0020h, l = 0x1000, f = 0x0FFFF
}
```


You normally use the MEMORY directive in conjunction with the SECTIONS directive to control allocation of output sections. After you use MEMORY to specify the target system's memory model, you can use SECTIONS to allocate output sections into specific named memory ranges or into memory that has specific attributes. For example, you could allocate the .text and .data sections into the area named FLASH and allocate the .bss section into the area named RAM.

7.8 The SECTIONS Directive

The SECTIONS directive controls your sections in the following ways:

- Describes how input sections are combined into output sections
- Defines output sections in the executable program
- Specifies where output sections are placed in memory (in relation to each other and to the entire memory space)
- Permits renaming of output sections

For more information, see [Section 2.3](#), [Section 2.4](#), and [Section 2.2.4](#). Subsections allow you to manipulate sections with greater precision.

If you do not specify a SECTIONS directive, the linker uses a default algorithm for combining and allocating the sections. [Section 7.12](#), describes this algorithm in detail.

7.8.1 SECTIONS Directive Syntax

The SECTIONS directive is specified in a command file by the word SECTIONS (uppercase), followed by a list of output section specifications enclosed in braces.

The general syntax for the SECTIONS directive is:

```
SECTIONS
{
    name : [property [, property] [, property] . . . ]
    name : [property [, property] [, property] . . . ]
    name : [property [, property] [, property] . . . ]
}
```

Each section specification, beginning with *name*, defines an output section. (An output section is a section in the output file.) A section name can be a subsection specification. (See [Section 7.8.4](#) for information on multi-level subsections.) After the section name is a list of properties that define the section's contents and how the section is allocated. The properties can be separated by optional commas. Possible properties for a section are as follows:

- **Load allocation** defines where in memory the section is to be loaded.
 Syntax: **load** = *allocation* or
 allocation or
 > *allocation*
- **Run allocation** defines where in memory the section is to be run.
 Syntax: **run** = *allocation* or
 run > *allocation*

- **Input sections** defines the input sections (object files) that constitute the output section.

Syntax: { *input_sections* }

- **Section type** defines flags for special section types.

Syntax: **type = COPY** or

type = DSECT or

type = NOLOAD

See [Section 7.11](#).

- **Fill value** defines the value used to fill uninitialized holes.

Syntax: **fill = value** or

name : [*properties* =
 value]

See [Section 7.14](#).

[Example 7-4](#) shows a *SECTIONS* directive in a sample link command file.

Example 7-4. The *SECTIONS* Directive

```

/*****
/*      Sample command file with SECTIONS directive      */
/*****
file1.obj file2.obj          /* Input files */
--output_file=progr.out     /* Options   */

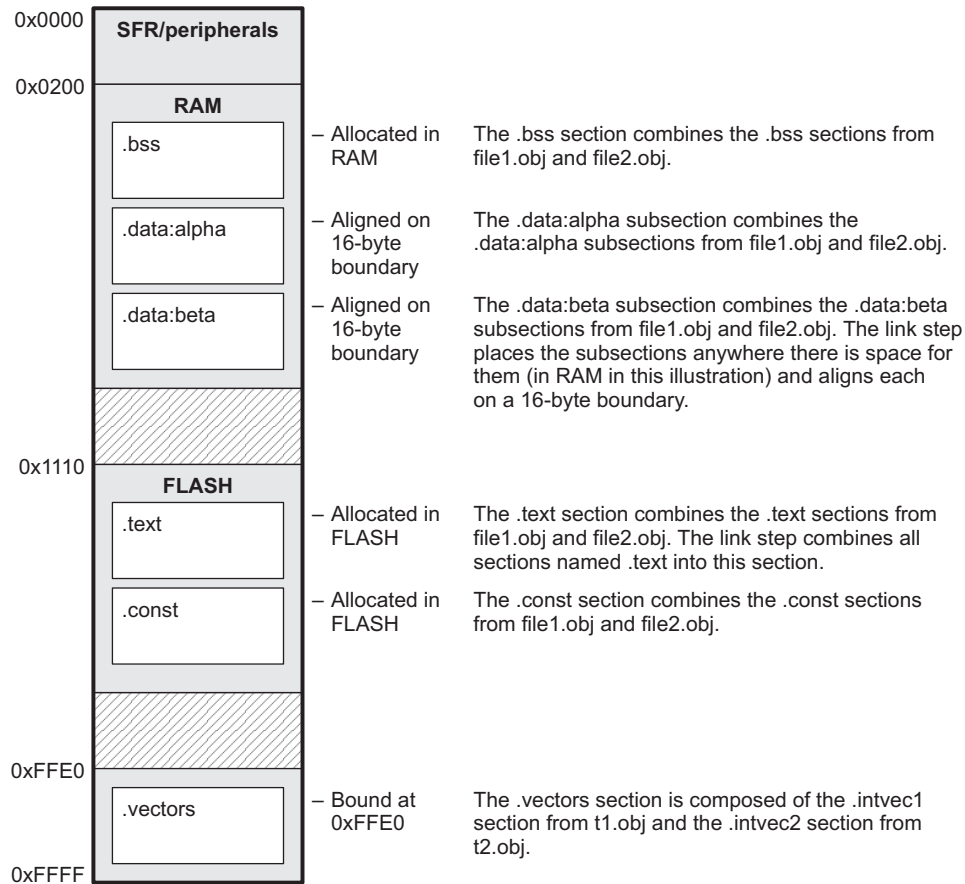
SECTIONS
{
    .bss      : load = RAM

    .text     : load = FLASH
    .const    : load = FLASH

    .vectors : load = 0xFFE0
    {
        t1.obj (.intvec1)
        t2.obj (.intvec2)
    }
    .data:alpha : align = 16
    .data:beta  : align = 16
}
    
```

[Figure 7-2](#) shows the six output sections defined by the *SECTIONS* directive in [Example 7-4](#) (.vectors, .text, .const, .bss, .data:alpha, and .data:beta) and shows how these sections are allocated in memory.

Figure 7-2. Section Allocation Defined by Example 7-4



7.8.2 Allocation

The linker assigns each output section two locations in target memory: the location where the section will be loaded and the location where it will be run. Usually, these are the same, and you can think of each section as having only a single address. The process of locating the output section in the target's memory and assigning its address(es) is called allocation. For more information about using separate load and run allocation, see [Section 7.9](#).

If you do not tell the linker how a section is to be allocated, it uses a default algorithm to allocate the section. Generally, the linker puts sections wherever they fit into configured memory. You can override this default allocation for a section by defining it within a SECTIONS directive and providing instructions on how to allocate it.

You control allocation by specifying one or more allocation parameters. Each parameter consists of a keyword, an optional equal sign or greater-than sign, and a value optionally enclosed in parentheses. If load and run allocation are separate, all parameters following the keyword LOAD apply to load allocation, and those following the keyword RUN apply to run allocation. The allocation parameters are:

Binding allocates a section at a specific address.

```
.text: load = 0x1000
```

Named memory allocates the section into a range defined in the MEMORY directive with the specified name (like SLOW_MEM) or attributes.

```
.text: load > SLOW_MEM
```

| | |
|------------------|---|
| Alignment | uses the <code>align</code> or <code>palign</code> keyword to specify that the section must start on an address boundary. <code>.text: align = 0x100</code> |
| Blocking | uses the <code>block</code> keyword to specify that the section must fit between two address boundaries: if the section is too big, it starts on an address boundary. <code>.text: block(0x100)</code> |

For the load (usually the only) allocation, you can simply use a greater-than sign and omit the load keyword:

```
text: > SLOW_MEM          .text: {...} > SLOW_MEM
.text: > 0x4000
```

If more than one parameter is used, you can string them together as follows:

```
.text: > SLOW_MEM align 16
```

Or if you prefer, use parentheses for readability:

```
.text: load = (SLOW_MEM align(16))
```

You can also use an input section specification to identify the sections from input files that are combined to form an output section. See [Section 7.8.3](#).

7.8.2.1 Binding

You can supply a specific starting address for an output section by following the section name with an address:

```
.text: 0x00001000
```

This example specifies that the `.text` section must begin at location `0x1000`. The binding address must be a 16-bit constant.

Output sections can be bound anywhere in configured memory (assuming there is enough space), but they cannot overlap. If there is not enough space to bind a section to a specified address, the linker issues an error message.

Binding is Incompatible With Alignment and Named Memory

Note: You cannot bind a section to an address if you use alignment or named memory. If you try to do this, the linker issues an error message.

7.8.2.2 Named Memory

You can allocate a section into a memory range that is defined by the `MEMORY` directive (see [Section 7.7](#)). This example names ranges and links sections into them:

```
MEMORY
{
    SLOW_MEM (RIX) : origin = 0x00000000, length = 0x00001000
    FAST_MEM (RWIX) : origin = 0x30000000, length = 0x00000300
}

SECTIONS
{
    .text : > SLOW_MEM
    .data : > FAST_MEM ALIGN(128)
    .bss : > FAST_MEM
}
```

In this example, the linker places `.text` into the area called `SLOW_MEM`. The `.data` and `.bss` output sections are allocated into `FAST_MEM`. You can align a section within a named memory range; the `.data` section is aligned on a 128-byte boundary within the `FAST_MEM` range.

Similarly, you can link a section into an area of memory that has particular attributes. To do this, specify a set of attributes (enclosed in parentheses) instead of a memory name. Using the same MEMORY directive declaration, you can specify:

```
SECTIONS
{
    .text: > (X) /* .text --> executable memory */
    .data: > (RI) /* .data --> read or init memory */
    .bss : > (RW) /* .bss --> read or write memory */
}
```

In this example, the .text output section can be linked into either the SLOW_MEM or FAST_MEM area because both areas have the X attribute. The .data section can also go into either SLOW_MEM or FAST_MEM because both areas have the R and I attributes. The .bss output section, however, must go into the FAST_MEM area because only FAST_MEM is declared with the W attribute.

You cannot control where in a named memory range a section is allocated, although the linker uses lower memory addresses first and avoids fragmentation when possible. In the preceding examples, assuming no conflicting assignments exist, the .text section starts at address 0. If a section must start on a specific address, use binding instead of named memory.

7.8.2.3 Controlling Allocation Using The HIGH Location Specifier

The linker allocates output sections from low to high addresses within a designated memory range by default. Alternatively, you can cause the linker to allocate a section from high to low addresses within a memory range by using the HIGH location specifier in the SECTION directive declaration.

For example, given this MEMORY directive:

```
MEMORY
{
    RAM          : origin = 0x0200, length = 0x0800
    FLASH       : origin = 0x1100, length = 0xEEE0
    VECTORS     : origin = 0xFFE0, length = 0x001E
    RESET      : origin = 0xFFFFE, length = 0x0002
}
```

and an accompanying SECTIONS directive:

```
SECTIONS
{
    .bss      : {} > RAM
    .system  : {} > RAM
    .stack   : {} > RAM (HIGH)
}
```

The HIGH specifier used on the .stack section allocation causes the linker to attempt to allocate .stack into the higher addresses within the RAM memory range. The .bss and .system sections are allocated into the lower addresses within RAM. [Example 7-5](#) illustrates a portion of a map file that shows where the given sections are allocated within RAM for a typical program.

Example 7-5. Linker Allocation With the HIGH Specifier

| | | | | | |
|---------|---|----------|----------|---------------|------------------------------------|
| .bss | 0 | 00000200 | 00000270 | UNINITIALIZED | |
| | | 00000200 | 0000011a | | rtsxxx.lib : defs.obj (.bss) |
| | | 0000031a | 00000088 | | : trgdrv.obj (.bss) |
| | | 000003a2 | 00000078 | | : lowlev.obj (.bss) |
| | | 0000041a | 00000046 | | : exit.obj (.bss) |
| | | 00000460 | 00000008 | | : memory.obj (.bss) |
| | | 00000468 | 00000004 | | : _lock.obj (.bss) |
| | | 0000046c | 00000002 | | : fopen.obj (.bss) |
| | | 0000046e | 00000002 | | hello.obj (.bss) |
| .system | 0 | 00000470 | 00000120 | UNINITIALIZED | |
| | | 00000470 | 00000004 | | rtsxxx .lib : memory.obj (.system) |
| .stack | 0 | 000008c0 | 00000140 | UNINITIALIZED | |
| | | 000008c0 | 00000002 | | rtsxxx .lib : boot.obj (.stack) |

As shown in [Example 7-5](#), the .bss and .system sections are allocated at the lower addresses of RAM (0x0200 - 0x0590) and the .stack section is allocated at address 0x08c0, even though lower addresses are available.

Without using the HIGH specifier, the linker allocation would result in the code shown in [Example 7-6](#)

The HIGH specifier is ignored if it is used with specific address binding or automatic section splitting (>> operator).

Example 7-6. Linker Allocation Without HIGH Specifier

| | | | | | |
|---------|---|----------|----------|---------------|------------------------|
| .bss | 0 | 00000200 | 00000270 | UNINITIALIZED | |
| | | 00000200 | 0000011a | rtsxxx.lib | : defs.obj (.bss) |
| | | 0000031a | 00000088 | | : trgdrv.obj (.bss) |
| | | 000003a2 | 00000078 | | : lowlev.obj (.bss) |
| | | 0000041a | 00000046 | | : exit.obj (.bss) |
| | | 00000460 | 00000008 | | : memory.obj (.bss) |
| | | 00000468 | 00000004 | | : _lock.obj (.bss) |
| | | 0000046c | 00000002 | | : fopen.obj (.bss) |
| | | 0000046e | 00000002 | hello.obj | (.bss) |
| .stack | 0 | 00000470 | 00000140 | UNINITIALIZED | |
| | | 00000470 | 00000002 | rtsxxx.lib | : boot.obj (.stack) |
| .system | 0 | 000005b0 | 00000120 | UNINITIALIZED | |
| | | 000005b0 | 00000004 | rtsxxx.lib | : memory.obj (.system) |

7.8.2.4 Alignment and Blocking

You can tell the linker to place an output section at an address that falls on an n-byte boundary, where n is a power of 2, by using the align keyword. For example, the following code allocates .text so that it falls on a 32-byte boundary:

```
.text: load = align(32)
```

You can specify the same alignment with the palign keyword. In addition, palign ensures the section's size is a multiple of its placement alignment restrictions, padding the section size up to such a boundary, as needed.

Blocking is a weaker form of alignment that allocates a section anywhere *within* a block of size n. The specified block size must be a power of 2. For example, the following code allocates .bss so that the entire section is contained in a single 128-byte page or begins on that boundary.:

```
bss: load = block(0x0080)
```

You can use alignment or blocking alone or in conjunction with a memory area, but alignment and blocking cannot be used together.

7.8.2.5 Alignment With Padding

As with align, you can tell the linker to place an output section at an address that falls on an n-byte boundary, where n is a power of 2, by using the palign keyword. In addition, palign ensures that the size of the section is a multiple of its placement alignment restrictions, padding the section size up to such a boundary, as needed.

For example, the following code lines allocate .text on a 2-byte boundary within the PMEM area. The .text section size is guaranteed to be a multiple of 2 bytes. Both statements are equivalent:

```
.text: palign(2) {} > PMEM

.text: palign = 2 {} > PMEM
```

If the linker adds padding to an initialized output section then the padding space is also initialized. By default, padding space is filled with a value of 0 (zero). However, if a fill value is specified for the output section then any padding for the section is also filled with that fill value.

For example, consider the following section specification:

```
.mytext: palign(8), fill = 0xffffffff {} > PMEM
```

In this example, the length of the `.mytext` section is 6 bytes before the `palign` operator is applied. The contents of `.mytext` are as follows:

```
addr content
----
0000 0x1234
0002 0x1234
0004 0x1234
```

After the `palign` operator is applied, the length of `.mytext` is 8 bytes, and its contents are as follows:

```
addr content
----
0000 0x1234
0002 0x1234
0004 0x1234
0006 0xffff
```

The size of `.mytext` has been bumped to a multiple of 8 bytes and the padding created by the linker has been filled with `0xff`.

The fill value specified in the linker command file is interpreted as a 16-bit constant, so if you specify this code:

```
.mytext: palign(8), fill = 0xff {} > PMEM
```

The fill value assumed by the linker is `0x00ff`, and `.mytext` will then have the following contents:

```
addr content
----
0000 0x1234
0002 0x1234
0004 0x1234
0006 0xffff
0008 0x00ff
000a 0x00ff
```

If the `palign` operator is applied to an uninitialized section, then the size of the section is bumped to the appropriate boundary, as needed, but any padding created is not initialized.

7.8.3 Specifying Input Sections

An input section specification identifies the sections from input files that are combined to form an output section. In general, the linker combines input sections by concatenating them in the order in which they are specified. However, if alignment or blocking is specified for an input section, all of the input sections within the output section are ordered as follows:

- All aligned sections, from largest to smallest
- All blocked sections, from largest to smallest
- All other sections, from largest to smallest

The size of an output section is the sum of the sizes of the input sections that it comprises.

[Example 7-7](#) shows the most common type of section specification; note that no input sections are listed.

Example 7-7. The Most Common Method of Specifying Section Contents

```
SECTIONS
{
    .text:
    .data:
    .bss:
}
```

In [Example 7-7](#), the linker takes all the `.text` sections from the input files and combines them into the `.text` output section. The linker concatenates the `.text` input sections in the order that it encounters them in the input files. The linker performs similar operations with the `.data` and `.bss` sections. You can use this type of specification for any output section.

You can explicitly specify the input sections that form an output section. Each input section is identified by its filename and section name:

```
SECTIONS
{
    .text :          /* Build .text output section      */
    {
        f1.obj(.text) /* Link .text section from f1.obj      */
        f2.obj(sec1) /* Link sec1 section from f2.obj      */
        f3.obj       /* Link ALL sections from f3.obj      */
        f4.obj(.text,sec2) /* Link .text and sec2 from f4.obj  */
    }
}
```

It is not necessary for input sections to have the same name as each other or as the output section they become part of. If a file is listed with no sections, *all* of its sections are included in the output section. If any additional input sections have the same name as an output section but are not explicitly specified by the SECTIONS directive, they are automatically linked in at the end of the output section. For example, if the linker found more `.text` sections in the preceding example and these `.text` sections *were not* specified anywhere in the SECTIONS directive, the linker would concatenate these extra sections after `f4.obj(sec2)`.

The specifications in [Example 7-7](#) are actually a shorthand method for the following:

```
SECTIONS
{
    .text: { *(.text) }
    .data: { *(.data) }
    .bss: { *(.bss) }
}
```

The specification `*(.text)` means *the unallocated .text sections from all the input files*. This format is useful when:

- You want the output section to contain all input sections that have a specified name, but the output section name is different from the input sections' name.
- You want the linker to allocate the input sections before it processes additional input sections or commands within the braces.

The following example illustrates the two purposes above:

```
SECTIONS
{
    .text : {
        abc.obj(xqt)
        *(.text)
    }
    .data : {
        *(.data)
        fil.obj(table)
    }
}
```

In this example, the `.text` output section contains a named section `xqt` from file `abc.obj`, which is followed by all the `.text` input sections. The `.data` section contains all the `.data` input sections, followed by a named section `table` from the file `fil.obj`. This method includes all the unallocated sections. For example, if one of the `.text` input sections was already included in another output section when the linker encountered `*(.text)`, the linker could not include that first `.text` input section in the second output section.

7.8.4 Using Multi-Level Subsections

Originally, subsections were identified with the base section name and a subsection name separated by a colon. For example, A:B names a subsection of the base section A. In certain places in a link command file specifying a base name, such as A, selects the section A as well as any subsections of A, such as A:B or A:C.

This concept has been extended to include multiple levels of subsection naming. The original constraints are still true, but a name such as A:B can be used to specify a (sub)section of that name as well as any (multi-level) subsections beginning with that name, such as A:B:C, A:B:OTHER, etc. All the subsections of A:B are also subsections of A. A and A:B are supersections of A:B:C. Among a group of supersections of a subsection, the nearest supersection is the supersection with the longest name. Thus, among {A, A:B} the nearest supersection of A:B:C:D is A:B.

With multiple levels of subsections, the constraints are the following:

1. When specifying **input** sections within a file (or library unit) the section name selects an input section of the same name and any subsections of that name.
2. Input sections that are not explicitly allocated are allocated in an existing **output** section of the same name or in the nearest existing supersection of such an output section. An exception to this rule is that during a partial link (specified by the --relocatable linker option) a subsection is allocated only to an existing output section of the same name.
3. If no such output section described in 2) is defined, the input section is put in a **newly created output** section with the same name as the base name of the input section

Consider linking input sections with the following names:

| | | |
|----------------------|------------------------|--------------------|
| europa:north:norway | europa:central:france | europa:south:spain |
| europa:north:sweden | europa:central:germany | europa:south:italy |
| europa:north:finland | europa:central:denmark | europa:south:malta |
| europa:north:iceland | | |

This SECTIONS specification allocates the input sections as indicated in the comments:

```
SECTIONS {
  nordic: {*(europa:north)
           *(europa:central:denmark)} /* the nordic countries */
  central: {*(europa:central)} /* france, germany */
  therest: {*(europa)} /* spain, italy, malta */
}
```

This SECTIONS specification allocates the input sections as indicated in the comments:

```
SECTIONS {
  islands: {*(europa:south:malta)
            *(europa:north:iceland)} /* malta, iceland */
  europa:north:finland : {} /* finland */
  europa:north : {} /* norway, sweden */
  europa:central : {} /* germany, denmark */
  europa:central:france: {} /* france */

  /* (italy, spain) go into a linker-generated output section "europa" */
}
```

Upward Compatibility of Multi-Level Subsections

Note: Existing linker commands that use the existing single-level subsection features and which do not contain section names containing multiple colon characters continue to behave as before. However, if section names in a link command file or in the input sections supplied to the linker contain multiple colon characters, some change in behavior could be possible. You should carefully consider the impact of the new rules for multiple levels to see if it affects a particular system link.

7.8.5 Allocation Using Multiple Memory Ranges

The linker allows you to specify an explicit list of memory ranges into which an output section can be allocated. Consider the following example:

```
MEMORY
{
    P_MEM1 : origin = 0x02000, length = 0x01000
    P_MEM2 : origin = 0x04000, length = 0x01000
    P_MEM3 : origin = 0x06000, length = 0x01000
    P_MEM4 : origin = 0x08000, length = 0x01000
}

SECTIONS
{
    .text : { } > P_MEM1 | P_MEM2 | P_MEM4
}
```

The | operator is used to specify the multiple memory ranges. The .text output section is allocated as a whole into the first memory range in which it fits. The memory ranges are accessed in the order specified. In this example, the linker first tries to allocate the section in P_MEM1. If that attempt fails, the linker tries to place the section into P_MEM2, and so on. If the output section is not successfully allocated in any of the named memory ranges, the linker issues an error message.

With this type of SECTIONS directive specification, the linker can seamlessly handle an output section that grows beyond the available space of the memory range in which it is originally allocated. Instead of modifying the link command file, you can let the linker move the section into one of the other areas.

7.8.6 Automatic Splitting of Output Sections Among Non-Contiguous Memory Ranges

The linker can split output sections among multiple memory ranges to achieve an efficient allocation. Use the >> operator to indicate that an output section can be split, if necessary, into the specified memory ranges. For example:

```
MEMORY
{
    P_MEM1 : origin = 0x2000, length = 0x1000
    P_MEM2 : origin = 0x4000, length = 0x1000
    P_MEM3 : origin = 0x6000, length = 0x1000
    P_MEM4 : origin = 0x8000, length = 0x1000
}

SECTIONS
{
    .text: { *(.text) } >> P_MEM1 | P_MEM2 | P_MEM3 | P_MEM4
}
```

In this example, the >> operator indicates that the .text output section can be split among any of the listed memory areas. If the .text section grows beyond the available memory in P_MEM1, it is split on an input section boundary, and the remainder of the output section is allocated to P_MEM2 | P_MEM3 | P_MEM4.

The | operator is used to specify the list of multiple memory ranges.

You can also use the >> operator to indicate that an output section can be split within a single memory range. This functionality is useful when several output sections must be allocated into the same memory range, but the restrictions of one output section cause the memory range to be partitioned. Consider the following example:

```
MEMORY
{
    RAM : origin = 0x1000, length = 0x8000
}

SECTIONS
{
    .special: { f1.obj(.text) } = 0x4000
    .text: { *(.text) } >> RAM
}
```

The `.special` output section is allocated near the middle of the RAM memory range. This leaves two unused areas in RAM: from `0x1000` to `0x4000`, and from the end of `f1.obj(.text)` to `0x8000`. The specification for the `.text` section allows the linker to split the `.text` section around the `.special` section and use the available space in RAM on either side of `.special`.

The `>>` operator can also be used to split an output section among all memory ranges that match a specified attribute combination. For example:

```
MEMORY
{
  P_MEM1 (RWX) : origin = 0x1000, length = 0x2000
  P_MEM2 (RWI) : origin = 0x4000, length = 0x1000
}

SECTIONS
{
  .text: { *(.text) } >> (RW)
}
```

The linker attempts to allocate all or part of the output section into any memory range whose attributes match the attributes specified in the SECTIONS directive.

This SECTIONS directive has the same effect as:

```
SECTIONS
{
  .text: { *(.text) } >> P_MEM1 | P_MEM2}
}
```

Certain sections should not be split:

- Certain sections created by the compiler, including
 - The `.cinit` section, which contains the autoinitialization table for C/C++ programs
 - The `.pinit` section, which contains the list of global constructors for C++ programs
 - The `.bss` section, which defines global variables
- An output section with an input section specification that includes an expression to be evaluated. The expression may define a symbol that is used in the program to manage the output section at run time.
- An output section that has a `START()`, `END()`, or `SIZE()` operator applied to it. These operators provide information about a section's load or run address, and size. Splitting the section may compromise the integrity of the operation.
- The run allocation of a UNION. (Splitting the load allocation of a UNION is allowed.)

If you use the `>>` operator on any of these sections, the linker issues a warning and ignores the operator.

7.8.7 Allocating an Archive Member to an Output Section

The ability to specify an archive member of a library archive for allocation into a specific output section can be specified inside angle brackets after a library name. Any object files separated by commas or spaces from the specified archive file are legal within the angle brackets. The syntax for allocating archived library members specifically inside of a SECTIONS directive is as follows:

```
[--library=] library name <member1[, member2[,] ...> [(input sections)]
```

Example 7-8 specifies that the text sections of `boot.obj`, `exit.obj`, and `strcpy.obj` from the run-time-support library should be placed in section `.boot`. The remainder of the `.text` sections from the run-time-support library are to be placed in section `.rts`. Finally, the remainder of all other `.text` sections are to be placed in section `.text`.

Example 7-8. Archive Members to Output Sections

```

SECTIONS
{
    boot    >      BOOT1
    {
        --library=rtsXX.lib<boot.obj> (.text)
        --library=rtsXX.lib<exit.obj strcpy.obj> (.text)
    }

    .rts    >      BOOT2
    {
        --library=rtsXX.lib (.text)
    }

    .text   >      RAM
    {
        * (.text)
    }
}
    
```

The `--library` option (which normally implies a library path search be made for the named file following the option) listed before each library in [Example 7-8](#) is optional when listing specific archive members inside `< >`. Using `< >` implies that you are referring to a library.

To collect a set of the input sections from a library in one place, use the `--library` option within the `SECTIONS` directive. For example, the following collects all the `.text` sections from `rts430.lib` into the `.rtstest` section:

```

SECTIONS
{
    .rtstest { --library=rts430.lib(.text) } > RAM
}
    
```

SECTIONS Directive Effect on `--priority`

Note: Specifying a library in a `SECTIONS` directive causes that library to be entered in the list of libraries that the linker searches to resolve references. If you use the `--priority` option, the first library specified in the command file will be searched first.

7.9 Specifying a Section's Run-Time Address

At times, you may want to load code into one area of memory and run it in another. For example, you may have performance-critical code in slow external memory. The code must be loaded into slow external memory, but it would run faster in fast external memory.

The linker provides a simple way to accomplish this. You can use the `SECTIONS` directive to direct the linker to allocate a section twice: once to set its load address and again to set its run address. For example:

```
.fir: load = SLOW_MEM, run = FAST_MEM
```

Use the *load* keyword for the load address and the *run* keyword for the run address.

See [Section 2.5](#) for an overview on run-time relocation.

7.9.1 Specifying Load and Run Addresses

The load address determines where a loader places the raw data for the section. Any references to the section (such as labels in it) refer to its run address. The application must copy the section from its load address to its run address; this does *not* happen automatically when you specify a separate run address.

If you provide only one allocation (either load or run) for a section, the section is allocated only once and loads and runs at the same address. If you provide both allocations, the section is allocated as if it were two sections of the same size. This means that both allocations occupy space in the memory map and cannot overlay each other or other sections. (The `UNION` directive provides a way to overlay sections; see [Section 7.10.1](#).)

If either the load or run address has additional parameters, such as alignment or blocking, list them after the appropriate keyword. Everything related to allocation after the keyword *load* affects the load address until the keyword *run* is seen, after which, everything affects the run address. The load and run allocations are completely independent, so any qualification of one (such as alignment) has no effect on the other. You can also specify run first, then load. Use parentheses to improve readability.

The examples below specify load and run addresses:

```
.data: load = SLOW_MEM, align = 32, run = FAST_MEM
```

(align applies only to load)

```
.data: load = (SLOW_MEM align 32), run = FAST_MEM
```

(identical to previous example)

```
.data: run = FAST_MEM, align 32,
      load = align 16
```

(align 32 in FAST_MEM for run; align 16 anywhere for load)

7.9.2 Uninitialized Sections

Uninitialized sections (such as `.bss`) are not loaded, so their only significant address is the run address. The linker allocates uninitialized sections only once: if you specify both run and load addresses, the linker warns you and ignores the load address. Otherwise, if you specify only one address, the linker treats it as a run address, regardless of whether you call it load or run. This example specifies load and run addresses for an uninitialized section:

```
.bss: load = 0x1000, run = FAST_MEM
```

A warning is issued, load is ignored, and space is allocated in FAST_MEM. All of the following examples have the same effect. The `.bss` section is allocated in FAST_MEM.

```
.bss: load = FAST_MEM
.bss: run = FAST_MEM
.bss: > FAST_MEM
```

7.9.3 Referring to the Load Address by Using the .label Directive

Normally, any reference to a symbol in a section refers to its run-time address. However, it may be necessary at run time to refer to a load-time address. Specifically, the code that copies a section from its load address to its run address must have access to the load address. The .label directive defines a special symbol that refers to the section's load address. Thus, whereas normal symbols are relocated with respect to the run address, .label symbols are relocated with respect to the load address. See [Create a Load-Time Address Label](#) for more information on the .label directive.

[Example 7-9](#) and [Example 7-10](#) show the use of the .label directive to copy a section from its load address in EXT_MEM to its run address in P_MEM. [Figure 7-3](#) illustrates the run-time execution of [Example 7-9](#).

Example 7-9. Copying Section Assembly Language File

```

* Define a section to be copied from load to run address
.sect ".fir"
.label fir_src
fir:
*   < code here>           ;code for section

        .label fir_end

* Copy .fir section from load address to run address
.text

        MOV    &fir_s,R11
        MOV    &fir_e,R12
        MOV    #fir,R13
LOOP:   CMP    R11,R12
        JL     Copy_Done
        MOV    @R11+,0(R13)
        INC   R13
        JMP   LOOP
Copy_Done:

* Jump to fir routine, now at run address

        JMP   fir

fir_s   .word  fir_src
fir_e   .word  fir_end
    
```

Example 7-10. Linker Command File for Example 7-9

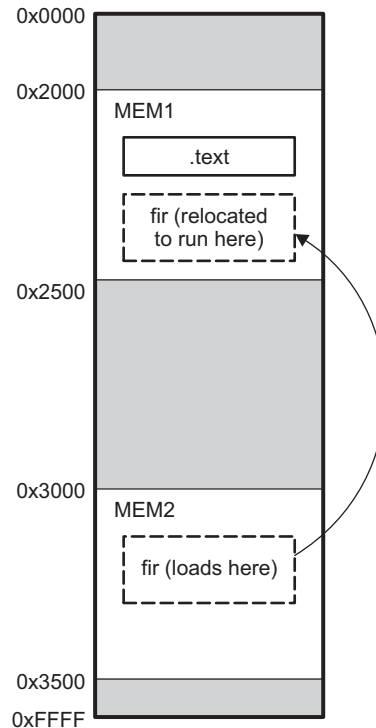
```

/*****
/*   PARTIAL LINKER COMMAND FILE FOR FIR EXAMPLE   */
*****/

MEMORY
{
    MEM1: origin = 0x2000, length 0x0500
    MEM2: origin = 0x3000, length 0x0500
}

SECTIONS
{
    .text: load = MEM1
    .fir:  load = MEM2, run = MEM1
}
    
```

Figure 7-3. Run-Time Execution of Example 7-9



7.10 Using UNION and GROUP Statements

Two SECTIONS statements allow you to conserve memory: GROUP and UNION. Unioning sections causes the linker to allocate them to the same run address. Grouping sections causes the linker to allocate them contiguously in memory. Section names can refer to sections, subsections, or archive library members.

7.10.1 Overlaying Sections With the UNION Statement

For some applications, you may want to allocate more than one section to run at the same address. For example, you may have several routines you want in fast external memory at various stages of execution. Or you may want several data objects that are not active at the same time to share a block of memory. The UNION statement within the SECTIONS directive provides a way to allocate several sections at the same run-time address.

In Example 7-11, the `.bss` sections from `file1.obj` and `file2.obj` are allocated at the same address in `FAST_MEM`. In the memory map, the union occupies as much space as its largest component. The components of a union remain independent sections; they are simply allocated together as a unit.

Example 7-11. The UNION Statement

```
SECTIONS
{
    .text: load = SLOW_MEM
    UNION: run = FAST_MEM
    {
        .bss:part1: { file1.obj(.bss) }
        .bss:part2: { file2.obj(.bss) }
    }
    .bss:part3: run = FAST_MEM { globals.obj(.bss) }
}
```

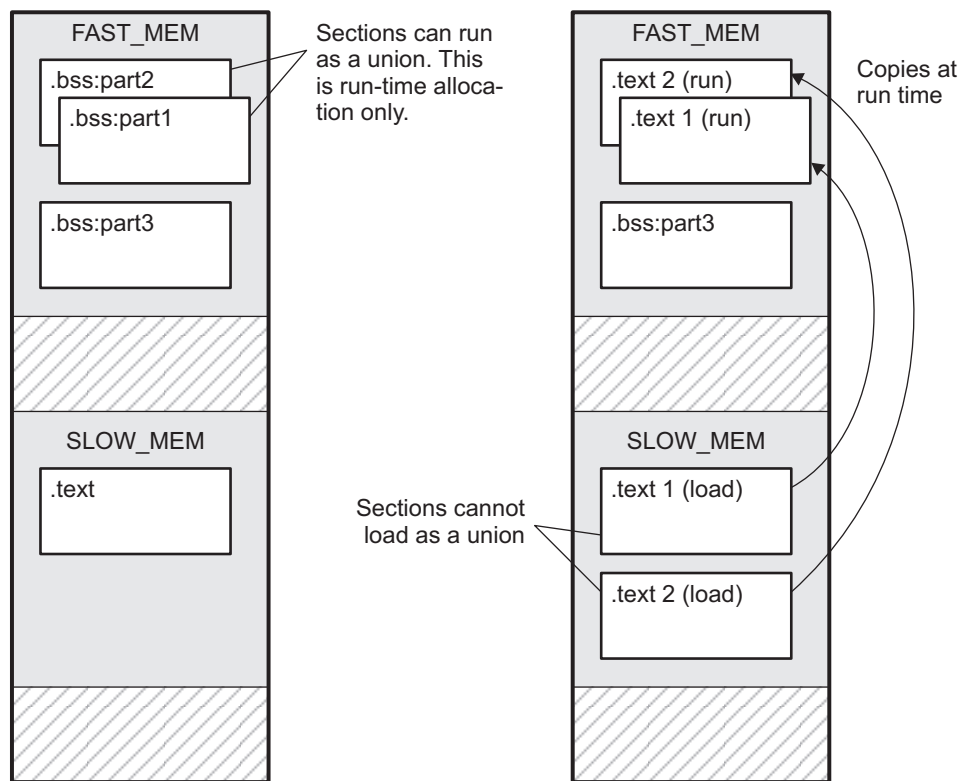
Allocation of a section as part of a union affects only its *run* address. Under no circumstances can sections be overlaid for loading. If an initialized section is a union member (an initialized section, such as `.text`, has raw data), its load allocation *must* be separately specified. See [Example 7-12](#).

Example 7-12. Separate Load Addresses for UNION Sections

```

UNION run = FAST_MEM
{
    .text:part1: load = SLOW_MEM, { file1.obj(.text) }
    .text:part2: load = SLOW_MEM, { file2.obj(.text) }
}
    
```

Figure 7-4. Memory Allocation Shown in [Example 7-11](#) and [Example 7-12](#)



Since the `.text` sections contain data, they cannot *load* as a union, although they can be *run* as a union. Therefore, each requires its own load address. If you fail to provide a load allocation for an initialized section within a UNION, the linker issues a warning and allocates load space anywhere it can in configured memory.

Uninitialized sections are not loaded and do not require load addresses.

The UNION statement applies only to allocation of run addresses, so it is meaningless to specify a load address for the union itself. For purposes of allocation, the union is treated as an uninitialized section: any one allocation specified is considered a run address, and if both run and load addresses are specified, the linker issues a warning and ignores the load address.

7.10.2 Grouping Output Sections Together

The SECTIONS directive's GROUP option forces several output sections to be allocated contiguously. For example, assume that a section named `term_rec` contains a termination record for a table in the `.data` section. You can force the linker to allocate `.data` and `term_rec` together:

Example 7-13. Allocate Sections Together

```

SECTIONS
{
    .text          /* Normal output section          */
    .bss           /* Normal output section          */
    GROUP 0x00001000 : /* Specify a group of sections */
    {
        .data      /* First section in the group     */
        term_rec   /* Allocated immediately after .data */
    }
}

```

You can use binding, alignment, or named memory to allocate a GROUP in the same manner as a single output section. In the preceding example, the GROUP is bound to address 0x1000. This means that .data is allocated at 0x1000, and term_rec follows it in memory.

You Cannot Specify Addresses for Sections Within a GROUP

Note: When you use the GROUP option, binding, alignment, or allocation into named memory can be specified for the group only. You cannot use binding, named memory, or alignment for sections within a group.

7.10.3 Nesting UNIONS and GROUPS

The linker allows arbitrary nesting of GROUP and UNION statements with the SECTIONS directive. By nesting GROUP and UNION statements, you can express hierarchical overlays and groupings of sections. [Example 7-14](#) shows how two overlays can be grouped together.

Example 7-14. Nesting GROUP and UNION Statements

```

SECTIONS
{
    GROUP 0x1000 : run = FAST_MEM
    {
        UNION:
        {
            mysect1: load = SLOW_MEM
            mysect2: load = SLOW_MEM
        }
        UNION:
        {
            mysect3: load = SLOW_MEM
            mysect4: load = SLOW_MEM
        }
    }
}

```

For this example, the linker performs the following allocations:

- The four sections (mysect1, mysect2, mysect3, mysect4) are assigned unique, non-overlapping load addresses. The name you defined is used in the SLOW_MEM memory region. This assignment is determined by the particular load allocations given for each section.
- Sections mysect1 and mysect2 are assigned the same run address in FAST_MEM.
- Sections mysect3 and mysect4 are assigned the same run address in FAST_MEM.
- The run addresses of mysect1/mysect2 and mysect3/mysect4 are allocated contiguously, as directed by the GROUP statement (subject to alignment and blocking restrictions).

To refer to groups and unions, linker diagnostic messages use the notation:

GROUP_n UNION_n

In this notation, n is a sequential number (beginning at 1) that represents the lexical ordering of the group or union in the linker control file, without regard to nesting. Groups and unions each have their own counter.

7.10.4 Checking the Consistency of Allocators

The linker checks the consistency of load and run allocations specified for unions, groups, and sections. The following rules are used:

- Run allocations are only allowed for top-level sections, groups, or unions (sections, groups, or unions that are not nested under any other groups or unions). The linker uses the run address of the top-level structure to compute the run addresses of the components within groups and unions.
- The linker does not accept a load allocation for UNIONS.
- The linker does not accept a load allocation for uninitialized sections.
- In most cases, you must provide a load allocation for an initialized section. However, the linker does not accept a load allocation for an initialized section that is located within a group that already defines a load allocator.
- As a shortcut, you can specify a load allocation for an entire group, to determine the load allocations for every initialized section or subgroup nested within the group. However, a load allocation is accepted for an entire group only if all of the following conditions are true:
 - The group is initialized (that is, it has at least one initialized member).
 - The group is not nested inside another group that has a load allocator.
 - The group does not contain a union containing initialized sections.
- If the group contains a union with initialized sections, it is necessary to specify the load allocation for each initialized section nested within the group. Consider the following example:

```
SECTIONS
{
  GROUP: load = SLOW_MEM, run = SLOW_MEM
  {
    .text1:
    UNION:
    {
      .text2:
      .text3:
    }
  }
}
```

- The load allocator given for the group does not uniquely specify the load allocation for the elements within the union: `.text2` and `.text3`. In this case, the linker issues a diagnostic message to request that these load allocations be specified explicitly.

7.10.5 Naming UNIONS and GROUPS

You can give a name to a UNION or GROUP by entering the name in parentheses after the declaration. For example:

```
GROUP(BSS_SYSMEM_STACK_GROUP)
{
  .bss :{}
  .sysmem :{}
  .stack :{}
} load=D_MEM, run=D_MEM
```

The name you defined is used in diagnostics for easy identification of the problem LCF area. For example:

```
warning: LOAD placement ignored for "BSS_SYSMEM_STACK_GROUP": object is uninitialized
```

```
UNION(TEXT_CINIT_UNION)
{
  .const :{}load=D_MEM, table(table1)
  .pinit :{}load=D_MEM, table(table1)
}run=P_MEM
```

```
warning:table(table1) operator ignored: table(table1) has already been applied to a section
in the "UNION(TEXT_CINIT_UNION)" in which ".pinit" is a descendant
```

7.11 Special Section Types (DSECT, COPY, and NOLOAD)

You can assign three special types to output sections: DSECT, COPY, and NOLOAD. These types affect the way that the program is treated when it is linked and loaded. You can assign a type to a section by placing the type after the section definition. For example:

```
SECTIONS
{
  sec1: load = 0x00002000, type = DSECT {f1.obj}
  sec2: load = 0x00004000, type = COPY {f2.obj}
  sec3: load = 0x00006000, type = NOLOAD {f3.obj}
}
```

- The DSECT type creates a dummy section with the following characteristics:
 - It is not included in the output section memory allocation. It takes up no memory and is not included in the memory map listing.
 - It can overlay other output sections, other DSECTs, and unconfigured memory.
 - Global symbols defined in a dummy section are relocated normally. They appear in the output module's symbol table with the same value they would have if the DSECT had actually been loaded. These symbols can be referenced by other input sections.
 - Undefined external symbols found in a DSECT cause specified archive libraries to be searched.
 - The section's contents, relocation information, and line number information are not placed in the output module.

In the preceding example, none of the sections from f1.obj are allocated, but all the symbols are relocated as though the sections were linked at address 0x2000. The other sections can refer to any of the global symbols in sec1.

- A COPY section is similar to a DSECT section, except that its contents and associated information are written to the output module. The .cinit section that contains initialization tables for the MSP430 C/C++ compiler has this attribute under the run-time initialization model.
- A NOLOAD section differs from a normal output section in one respect: the section's contents, relocation information, and line number information are not placed in the output module. The linker allocates space for the section, and it appears in the memory map listing.

7.12 Default Allocation Algorithm

The MEMORY and SECTIONS directives provide flexible methods for building, combining, and allocating sections. However, any memory locations or sections that you choose *not* to specify must still be handled by the linker. The linker uses default algorithms to build and allocate sections within the specifications you supply.

If you do not use the MEMORY and SECTIONS directives, the linker allocates output sections as though the definitions in [Example 7-15](#) were specified.

Example 7-15. Default Allocation for MSP430 Devices

```
MEMORY
{
  MEM      : origin = 0x0200, length = 0xFDFD
  RESET    : origin = 0xFFFFE, length = 0x0002
}

SECTIONS
{
  .text    : {} > MEM
  .const   : {} > MEM
  .data    : {} > MEM
  .bss     : {} > MEM

  .reset   :      > RESET

  .cinit   : {} > MEM      ;cflag option only
  .pinit   : {} > MEM      ;cflag option only
}
```

All .text input sections are concatenated to form a .text output section in the executable output file, and all .data input sections are combined to form a .data output section.

If you use a SECTIONS directive, the linker performs *no part* of the default allocation. Allocation is performed according to the rules specified by the SECTIONS directive and the general algorithm described next in [Section 7.12.1](#).

7.12.1 How the Allocation Algorithm Creates Output Sections

An output section can be formed in one of two ways:

Method 1 As the result of a SECTIONS directive definition

Method 2 By combining input sections with the same name into an output section that is not defined in a SECTIONS directive

If an output section is formed as a result of a SECTIONS directive, this definition completely determines the section's contents. (See [Section 7.8](#) for examples of how to define an output section's content.)

If an output section is formed by combining input sections not specified by a SECTIONS directive, the linker combines all such input sections that have the same name into an output section with that name. For example, suppose the files f1.obj and f2.obj both contain named sections called Vectors and that the SECTIONS directive does not define an output section for them. The linker combines the two Vectors sections from the input files into a single output section named Vectors, allocates it into memory, and includes it in the output file.

By default, the linker does not display a message when it creates an output section that is not defined in the SECTIONS directive. You can use the `--warn_sections` linker option (see [Section 7.4.35](#)) to cause the linker to display a message when it creates a new output section.

After the linker determines the composition of all output sections, it must allocate them into configured memory. The MEMORY directive specifies which portions of memory are configured. If there is no MEMORY directive, the linker uses the default configuration as shown in [Example 7-15](#). (See [Section 7.7](#) for more information on configuring memory.)

7.12.2 Reducing Memory Fragmentation

The linker's allocation algorithm attempts to minimize memory fragmentation. This allows memory to be used more efficiently and increases the probability that your program will fit into memory. The algorithm comprises these steps:

1. Each output section for which you have supplied a specific binding address is placed in memory at that address.
2. Each output section that is included in a specific, named memory range or that has memory attribute restrictions is allocated. Each output section is placed into the first available space within the named area, considering alignment where necessary.
3. Any remaining sections are allocated in the order in which they are defined. Sections not defined in a SECTIONS directive are allocated in the order in which they are encountered. Each output section is placed into the first available memory space, considering alignment where necessary.

7.13 Assigning Symbols at Link Time

Linker assignment statements allow you to define external (global) symbols and assign values to them at link time. You can use this feature to initialize a variable or pointer to an allocation-dependent value.

7.13.1 Syntax of Assignment Statements

The syntax of assignment statements in the linker is similar to that of assignment statements in the C language:

| | | | |
|---------------|----|---------------------|---|
| <i>symbol</i> | = | <i>expression</i> ; | assigns the value of expression to symbol |
| <i>symbol</i> | += | <i>expression</i> ; | adds the value of expression to symbol |
| <i>symbol</i> | -= | <i>expression</i> ; | subtracts the value of expression from symbol |
| <i>symbol</i> | *= | <i>expression</i> ; | multiplies symbol by expression |
| <i>symbol</i> | /= | <i>expression</i> ; | divides symbol by expression |

The symbol should be defined externally. If it is not, the linker defines a new symbol and enters it into the symbol table. The expression must follow the rules defined in [Section 7.13.3](#). Assignment statements *must* terminate with a semicolon.

The linker processes assignment statements *after* it allocates all the output sections. Therefore, if an expression contains a symbol, the address used for that symbol reflects the symbol's address in the executable output file.

For example, suppose a program reads data from one of two tables identified by two external symbols, Table1 and Table2. The program uses the symbol `cur_tab` as the address of the current table. The `cur_tab` symbol must point to either Table1 or Table2. You could accomplish this in the assembly code, but you would need to reassemble the program to change tables. Instead, you can use a linker assignment statement to assign `cur_tab` at link time:

```
prog.obj          /* Input file */
cur_tab = Table1; /* Assign cur_tab to one of the tables */
```

7.13.2 Assigning the SPC to a Symbol

A special symbol, denoted by a dot (`.`), represents the current value of the section program counter (SPC) during allocation. The SPC keeps track of the current location within a section. The linker's `.` symbol is analogous to the assembler's `$` symbol. The `.` symbol can be used only in assignment statements within a `SECTIONS` directive because `.` is meaningful only during allocation and `SECTIONS` controls the allocation process. (See [Section 7.8](#).)

The `.` symbol refers to the current run address, not the current load address, of the section.

For example, suppose a program needs to know the address of the beginning of the `.data` section. By using the `.global` directive (see [Identify Global Symbols](#)), you can create an external undefined variable called `Dstart` in the program. Then, assign the value of `.` to `Dstart`:

```
SECTIONS
{
  .text:    {}
  .data:    {Dstart = .;}
  .bss :    {}
}
```

This defines `Dstart` to be the first linked address of the `.data` section. (`Dstart` is assigned *before* `.data` is allocated.) The linker relocates all references to `Dstart`.

A special type of assignment assigns a value to the `.` symbol. This adjusts the SPC within an output section and creates a hole between two input sections. Any value assigned to `.` to create a hole is relative to the beginning of the section, not to the address actually represented by the `.` symbol. Holes and assignments to `.` are described in [Section 7.14](#).

7.13.3 Assignment Expressions

These rules apply to linker expressions:

- Expressions can contain global symbols, constants, and the C language operators listed in .
- All numbers are treated as long (32-bit) integers.
- Constants are identified by the linker in the same way as by the assembler. That is, numbers are recognized as decimal unless they have a suffix (H or h for hexadecimal and Q or q for octal). C language prefixes are also recognized (0 for octal and 0x for hex). Hexadecimal constants must begin with a digit. No binary constants are allowed.
- Symbols within an expression have only the value of the symbol's *address*. No type-checking is performed.
- Linker expressions can be absolute or relocatable. If an expression contains *any* relocatable symbols (and 0 or more constants or absolute symbols), it is relocatable. Otherwise, the expression is absolute. If a symbol is assigned the value of a relocatable expression, it is relocatable; if it is assigned the value of an absolute expression, it is absolute.

The linker supports the C language operators listed in [Table 7-2](#) in order of precedence. Operators in the same group have the same precedence. Besides the operators listed in , the linker also has an align operator that allows a symbol to be aligned on an n-byte boundary within an output section (n is a power of 2). For example, the following expression aligns the SPC within the current section on the next 16-byte boundary. Because the align operator is a function of the current SPC, it can be used only in the same context as . —that is, within a SECTIONS directive.

```
. = align(16);
```

Table 7-2. Groups of Operators Used in Expressions (Precedence)

| Group 1 (Highest Precedence) | | Group 6 | |
|------------------------------|--------------------------|------------------------------|--------------------|
| ! | Logical NOT | & | Bitwise AND |
| ~ | Bitwise NOT | | |
| - | Negation | | |
| Group 2 | | Group 7 | |
| * | Multiplication | | Bitwise OR |
| / | Division | | |
| % | Modulus | | |
| Group 3 | | Group 8 | |
| + | Addition | && | Logical AND |
| - | Subtraction | | |
| Group 4 | | Group 9 | |
| >> | Arithmetic right shift | | Logical OR |
| << | Arithmetic left shift | | |
| Group 5 | | Group 10 (Lowest Precedence) | |
| == | Equal to | = | Assignment |
| != | Not equal to | += | A += B " A = A + B |
| > | Greater than | -= | A -= B " A = A - B |
| < | Less than | *= | A *= B " A = A * B |
| <= | Less than or equal to | /= | A /= B " A = A / B |
| >= | Greater than or equal to | | |

7.13.4 Symbols Defined by the Linker

The linker automatically defines several symbols based on which sections are used in your assembly source. A program can use these symbols at run time to determine where a section is linked. Since these symbols are external, they appear in the linker map. Each symbol can be accessed in any assembly language module if it is declared with a `.global` directive (see [Identify Global Symbols](#)). You must have used the corresponding section in a source module for the symbol to be created. Values are assigned to these symbols as follows:

| | |
|--------------|---|
| .text | is assigned the first address of the <code>.text</code> output section. (It marks the <i>beginning</i> of executable code.) |
| etext | is assigned the first address following the <code>.text</code> output section. (It marks the <i>end</i> of executable code.) |
| .data | is assigned the first address of the <code>.data</code> output section. (It marks the <i>beginning</i> of initialized data tables.) |
| edata | is assigned the first address following the <code>.data</code> output section. (It marks the <i>end</i> of initialized data tables.) |
| .bss | is assigned the first address of the <code>.bss</code> output section. (It marks the <i>beginning</i> of uninitialized data.) |
| end | is assigned the first address following the <code>.bss</code> output section. (It marks the <i>end</i> of uninitialized data.) |

The following symbols are defined only for C/C++ support when the `--ram_model` or `--rom_model` option is used.

| | |
|----------------------|---|
| __STACK_SIZE | is assigned the size of the <code>.stack</code> section. |
| __SYSTEM_SIZE | is assigned the size of the <code>.sysmem</code> section. |

7.13.5 Assigning Exact Start, End, and Size Values of a Section to a Symbol

The code generation tools currently support the ability to load program code in one area of (slow) memory and run it in another (faster) area. This is done by specifying separate load and run addresses for an output section or group in the link command file. Then execute a sequence of instructions (the copying code in [Example 7-9](#)) that moves the program code from its load area to its run area before it is needed.

There are several responsibilities that a programmer must take on when setting up a system with this feature. One of these responsibilities is to determine the size and run-time address of the program code to be moved. The current mechanisms to do this involve use of the `.label` directives in the copying code. A simple example is illustrated [Example 7-9](#).

This method of specifying the size and load address of the program code has limitations. While it works fine for an individual input section that is contained entirely within one source file, this method becomes more complicated if the program code is spread over several source files or if the programmer wants to copy an entire output section from load space to run space.

Another problem with this method is that it does not account for the possibility that the section being moved may have an associated far call trampoline section that needs to be moved with it.

7.13.6 Why the Dot Operator Does Not Always Work

The dot operator (.) is used to define symbols at link-time with a particular address inside of an output section. It is interpreted like a PC. Whatever the current offset within the current section is, that is the value associated with the dot. Consider an output section specification within a SECTIONS directive:

```
outsect:
{
    s1.obj(.text)
    end_of_s1 = .;
    start_of_s2 = .;
    s2.obj(.text)
    end_of_s2 = .;
}
```

This statement creates three symbols:

- end_of_s1—the end address of .text in s1.obj
- start_of_s2—the start address of .text in s2.obj
- end_of_s2—the end address of .text in s2.obj

Suppose there is padding between s1.obj and s2.obj that is created as a result of alignment. Then start_of_s2 is not really the start address of the .text section in s2.obj, but it is the address before the padding needed to align the .text section in s2.obj. This is due to the linker's interpretation of the dot operator as the current PC. It is also due to the fact that the dot operator is evaluated independently of the input sections around it.

Another potential problem in the above example is that end_of_s2 may not account for any padding that was required at the end of the output section. You cannot reliably use end_of_s2 as the end address of the output section. One way to get around this problem is to create a dummy section immediately after the output section in question. For example:

```
GROUP
{
    outsect:
    {
        start_of_outsect = .;
        ...
    }
    dummy: { size_of_outsect = . - start_of_outsect; }
}
```

7.13.7 Address and Dimension Operators

Six new operators have been added to the link command file syntax:

| | |
|---|--|
| LOAD_START(sym) START(sym) | Defines <i>sym</i> with the load-time start address of related allocation unit |
| LOAD_END(sym) END(sym) | Defines <i>sym</i> with the load-time end address of related allocation unit |
| LOAD_SIZE(sym) SIZE(sym) | Defines <i>sym</i> with the load-time size of related allocation unit |
| RUN_START(sym) | Defines <i>sym</i> with the run-time start address of related allocation unit |
| RUN_END(sym) | Defines <i>sym</i> with the run-time end address of related allocation unit |
| RUN_SIZE(sym) | Defines <i>sym</i> with the run-time size of related allocation unit |

Linker Command File Operator Equivalencies

Note: LOAD_START() and START() are equivalent, as are LOAD_END()/END() and LOAD_SIZE()/SIZE().

The new address and dimension operators can be associated with several different kinds of allocation units, including input items, output sections, GROUPs, and UNIONs. The following sections provide some examples of how the operators can be used in each case.

7.13.7.1 Input Items

Consider an output section specification within a SECTIONS directive:

```
outsect:
{
    s1.obj(.text)
    end_of_s1 = .;
    start_of_s2 = .;
    s2.obj(.text)
    end_of_s2 = .;
}
```

This can be rewritten using the START and END operators as follows:

```
outsect:
{
    s1.obj(.text) { END(end_of_s1) }
    s2.obj(.text) { START(start_of_s2), END(end_of_s2) }
}
```

The values of end_of_s1 and end_of_s2 will be the same as if you had used the dot operator in the original example, but start_of_s2 would be defined after any necessary padding that needs to be added between the two .text sections. Remember that the dot operator would cause start_of_s2 to be defined before any necessary padding is inserted between the two input sections.

The syntax for using these operators in association with input sections calls for braces { } to enclose the operator list. The operators in the list are applied to the input item that occurs immediately before the list.

7.13.7.2 Output Section

The START, END, and SIZE operators can also be associated with an output section. Here is an example:

```
outsect: START(start_of_outsect), SIZE(size_of_outsect)
{
    <list of input items>
}
```

In this case, the SIZE operator defines size_of_outsect to incorporate any padding that is required in the output section to conform to any alignment requirements that are imposed.

The syntax for specifying the operators with an output section do not require braces to enclose the operator list. The operator list is simply included as part of the allocation specification for an output section.

7.13.7.3 GROUPs

Here is another use of the START and SIZE operators in the context of a GROUP specification:

```
GROUP
{
    outsect1: { ... }
    outsect2: { ... }
} load = ROM, run = RAM, START(group_start), SIZE(group_size);
```

This can be useful if the whole GROUP is to be loaded in one location and run in another. The copying code can use group_start and group_size as parameters for where to copy from and how much is to be copied. This makes the use of .label in the source code unnecessary.

7.13.7.4 UNIONS

The RUN_SIZE and LOAD_SIZE operators provide a mechanism to distinguish between the size of a UNION's load space and the size of the space where its constituents are going to be copied before they are run. Here is an example:

```
UNION: run = RAM, LOAD_START(union_load_addr),
      LOAD_SIZE(union_ld_sz), RUN_SIZE(union_run_sz)
{
    .text1: load = ROM, SIZE(text1_size) { f1.obj(.text) }
    .text2: load = ROM, SIZE(text2_size) { f2.obj(.text) }
}
```

Here union_ld_sz is going to be equal to the sum of the sizes of all output sections placed in the union. The union_run_sz value is equivalent to the largest output section in the union. Both of these symbols incorporate any padding due to blocking or alignment requirements.

7.14 Creating and Filling Holes

The linker provides you with the ability to create areas *within output sections* that have nothing linked into them. These areas are called *holes*. In special cases, uninitialized sections can also be treated as holes. This section describes how the linker handles holes and how you can fill holes (and uninitialized sections) with values.

7.14.1 Initialized and Uninitialized Sections

There are two rules to remember about the contents of output sections. An output section contains either:

- Raw data for the *entire* section
- No raw data

A section that has raw data is referred to as *initialized*. This means that the object file contains the actual memory image contents of the section. When the section is loaded, this image is loaded into memory at the section's specified starting address. The .text and .data sections *always* have raw data if anything was assembled into them. Named sections defined with the .sect assembler directive also have raw data.

By default, the .bss section (see [Reserve Space in the .bss Section](#)) and sections defined with the .usect directive (see [Reserve Uninitialized Space](#)) have no raw data (they are *uninitialized*). They occupy space in the memory map but have no actual contents. Uninitialized sections typically reserve space in fast external memory for variables. In the object file, an uninitialized section has a normal section header and can have symbols defined in it; no memory image, however, is stored in the section.

7.14.2 Creating Holes

You can create a hole in an initialized output section. A hole is created when you force the linker to leave extra space between input sections within an output section. When such a hole is created, *the linker must supply raw data for the hole*.

Holes can be created only *within* output sections. Space can exist *between* output sections, but such space is not a hole. To fill the space between output sections, see [Section 7.7.2](#).

To create a hole in an output section, you must use a special type of linker assignment statement within an output section definition. The assignment statement modifies the SPC (denoted by .) by adding to it, assigning a greater value to it, or aligning it on an address boundary. The operators, expressions, and syntaxes of assignment statements are described in [Section 7.13](#).

The following example uses assignment statements to create holes in output sections:

```
SECTIONS
{
  outsect:
  {
    file1.obj(.text)
    . += 0x0100 /* Create a hole with size 0x0100 */
    file2.obj(.text)
    . = align(16); /* Create a hole to align the SPC */
    file3.obj(.text)
  }
}
```

The output section outsect is built as follows:

1. The .text section from file1.obj is linked in.
2. The linker creates a 256-byte hole.
3. The .text section from file2.obj is linked in after the hole.
4. The linker creates another hole by aligning the SPC on a 16-byte boundary.
5. Finally, the .text section from file3.obj is linked in.

All values assigned to the `.` symbol within a section refer to the *relative address within the section*. The linker handles assignments to the `.` symbol as if the section started at address 0 (even if you have specified a binding address). Consider the statement `. = align(16)` in the example. This statement effectively aligns the file3.obj .text section to start on a 16-byte boundary within outsect. If outsect is ultimately allocated to start on an address that is not aligned, the file3.obj .text section will not be aligned either.

The `.` symbol refers to the current run address, not the current load address, of the section.

Expressions that decrement the `.` symbol are illegal. For example, it is invalid to use the `-=` operator in an assignment to the `.` symbol. The most common operators used in assignments to the `.` symbol are `+=` and `align`.

If an output section contains all input sections of a certain type (such as .text), you can use the following statements to create a hole at the beginning or end of the output section.

```
.text: { . += 0x0100; } /* Hole at the beginning */
.data: { *(.data)
        . += 0x0100; } /* Hole at the end */
```

Another way to create a hole in an output section is to combine an uninitialized section with an initialized section to form a single output section. *In this case, the linker treats the uninitialized section as a hole and supplies data for it.* The following example illustrates this method:

```
SECTIONS
{
  outsect:
  {
    file1.obj(.text)
    file1.obj(.bss) /* This becomes a hole */
  }
}
```

Because the .text section has raw data, all of outsect must also contain raw data. Therefore, the uninitialized .bss section becomes a hole.

Uninitialized sections become holes only when they are combined with initialized sections. If several uninitialized sections are linked together, the resulting output section is also uninitialized.

7.14.3 Filling Holes

When a hole exists in an initialized output section, the linker must supply raw data to fill it. The linker fills holes with a 32-bit fill value that is replicated through memory until it fills the hole. The linker determines the fill value as follows:

1. If the hole is formed by combining an uninitialized section with an initialized section, you can specify a fill value for the uninitialized section. Follow the section name with an = sign and a 32-bit constant. For example:

```
SECTIONS
{
  outsect:
  {
    file1.obj(.text)
    file2.obj(.bss)= 0xFF00FF00 /* Fill this hole with 0xFF00FF00 */
  }
}
```

2. You can also specify a fill value for all the holes in an output section by supplying the fill value after the section definition:

```
SECTIONS
{
  outsect:fill = 0xFF00FF00 /* Fills holes with 0xFF00FF00 */
  {
    . += 0x0010; /* This creates a hole */
    file1.obj(.text)
    file1.obj(.bss) /* This creates another hole */
  }
}
```

3. If you do not specify an initialization value for a hole, the linker fills the hole with the value specified with the `--fill_value` option (see [Section 7.4.10](#)). For example, suppose the command file `link.cmd` contains the following `SECTIONS` directive:

```
SECTIONS
{
  .text: { . = 0x0100; } /* Create a 100 word hole */
}
```

Now invoke the linker with the `--fill_value` option:

```
cl430 --run_linker --fill_value=0xFFFFFFFF link.cmd
```

This fills the hole with `0xFFFFFFFF`.

4. If you do not invoke the linker with the `--fill_value` option or otherwise specify a fill value, the linker fills holes with 0s.

Whenever a hole is created and filled in an initialized output section, the hole is identified in the link map along with the value the linker uses to fill it.

7.14.4 Explicit Initialization of Uninitialized Sections

You can force the linker to initialize an uninitialized section by specifying an explicit fill value for it in the `SECTIONS` directive. This causes the entire section to have raw data (the fill value). For example:

```
SECTIONS
{
  .bss: fill = 0x12341234 /* Fills .bss with 0x12341234 */
}
```

Filling Sections

Note: Because filling a section (even with 0s) causes raw data to be generated for the entire section in the output file, your output file will be very large if you specify fill values for large sections or holes.

7.15 Linker-Generated Copy Tables

The linker supports extensions to the link command file syntax that enable the following:

- Make it easier for you to copy objects from load-space to run-space at boot time
- Make it easier for you to manage memory overlays at run time
- Allow you to split GROUPs and output sections that have separate load and run addresses

7.15.1 A Current Boot-Loaded Application Development Process

In some embedded applications, there is a need to copy or download code and/or data from one location to another at boot time before the application actually begins its main execution thread. For example, an application may have its code and/or data in FLASH memory and need to copy it into on-chip memory before the application begins execution.

One way you can develop an application like this is to create a copy table in assembly code that contains three elements for each block of code or data that needs to be moved from FLASH into on-chip memory at boot time:

- The load address
- The run address
- The size

The process you follow to develop such an application might look like this:

1. Build the application to produce a .map file that contains the load and run addresses of each section that has a separate load and run placement.
2. Edit the copy table (used by the boot loader) to correct the load and run addresses as well as the size of each block of code or data that needs to be moved at boot time.
3. Build the application again, incorporating the updated copy table.
4. Run the application.

This process puts a heavy burden on you to maintain the copy table (by hand, no less). Each time a piece of code or data is added or removed from the application, you must repeat the process in order to keep the contents of the copy table up to date.

7.15.2 An Alternative Approach

You can avoid some of this maintenance burden by using the `LOAD_START()`, `RUN_START()`, and `SIZE()` operators that are already part of the link command file syntax. For example, instead of building the application to generate a .map file, the link command file can be annotated:

```
SECTIONS
{
    .flashcode: { app_tasks.obj(.text) }
        load = FLASH, run = PMEM,
        LOAD_START(_flash_code_ld_start),
        RUN_START(_flash_code_rn_start),
        SIZE(_flash_code_size)

    ...
}
```

In this example, the `LOAD_START()`, `RUN_START()`, and `SIZE()` operators instruct the linker to create three symbols:

| Symbol | Description |
|-----------------------------------|------------------------------------|
| <code>_flash_code_ld_start</code> | Load address of .flashcode section |
| <code>_flash_code_rn_start</code> | Run address of .flashcode section |
| <code>_flash_code_size</code> | Size of .flashcode section |

These symbols can then be referenced from the copy table. The actual data in the copy table will be updated automatically each time the application is linked. This approach removes step 1 of the process described in [Section 7.15.1](#).

While maintenance of the copy table is reduced markedly, you must still carry the burden of keeping the copy table contents in sync with the symbols that are defined in the link command file. Ideally, the linker would generate the boot copy table automatically. This would avoid having to build the application twice *and* free you from having to explicitly manage the contents of the boot copy table.

For more information on the `LOAD_START()`, `RUN_START()`, and `SIZE()` operators, see [Section 7.13.7](#).

7.15.3 Overlay Management Example

Consider an application which contains a memory overlay that must be managed at run time. The memory overlay is defined using a `UNION` in the link command file as illustrated in [Example 7-16](#):

Example 7-16. Using a `UNION` for Memory Overlay

```
SECTIONS
{
    ...

    UNION
    {
        GROUP
        {
            .task1: { task1.obj(.text) }
            .task2: { task2.obj(.text) }

        } load = ROM, LOAD_START(_task12_load_start), SIZE(_task12_size)

        GROUP
        {
            .task3: { task3.obj(.text) }
            .task4: { task4.obj(.text) }

        } load = ROM, LOAD_START(_task34_load_start), SIZE(_task_34_size)

    } run = RAM, RUN_START(_task_run_start)

    ...
}
```

The application must manage the contents of the memory overlay at run time. That is, whenever any services from `.task1` or `.task2` are needed, the application must first ensure that `.task1` and `.task2` are resident in the memory overlay. Similarly for `.task3` and `.task4`.

To affect a copy of `.task1` and `.task2` from ROM to RAM at run time, the application must first gain access to the load address of the tasks (`_task12_load_start`), the run address (`_task_run_start`), and the size (`_task12_size`). Then this information is used to perform the actual code copy.

7.15.4 Generating Copy Tables Automatically With the Linker

The linker supports extensions to the link command file syntax that enable you to do the following:

- Identify any object components that may need to be copied from load space to run space at some point during the run of an application
- Instruct the linker to automatically generate a copy table that contains (at least) the load address, run address, and size of the component that needs to be copied
- Instruct the linker to generate a symbol specified by you that provides the address of a linker-generated copy table. For instance, [Example 7-16](#) can be written as shown in [Example 7-17](#):

Example 7-17. Produce Address for Linker Generated Copy Table

```

SECTIONS
{
    ...

    UNION
    {
        GROUP
        {
            .task1: { task1.obj(.text) }
            .task2: { task2.obj(.text) }

        } load = ROM, table(_task12_copy_table)

        GROUP
        {
            .task3: { task3.obj(.text) }
            .task4: { task4.obj(.text) }

        } load = ROM, table(_task34_copy_table)

    } run = RAM
    ...
}
    
```

Using the SECTIONS directive from [Example 7-17](#) in the link command file, the linker generates two copy tables named: `_task12_copy_table` and `_task34_copy_table`. Each copy table provides the load address, run address, and size of the GROUP that is associated with the copy table. This information is accessible from application source code using the linker-generated symbols, `_task12_copy_table` and `_task34_copy_table`, which provide the addresses of the two copy tables, respectively.

Using this method, you do not have to worry about the creation or maintenance of a copy table. You can reference the address of any copy table generated by the linker in C/C++ or assembly source code, passing that value to a general purpose copy routine which will process the copy table and affect the actual copy.

7.15.5 The `table()` Operator

You can use the `table()` operator to instruct the linker to produce a copy table. A `table()` operator can be applied to an output section, a GROUP, or a UNION member. The copy table generated for a particular `table()` specification can be accessed through a symbol specified by you that is provided as an argument to the `table()` operator. The linker creates a symbol with this name and assigns it the address of the copy table as the value of the symbol. The copy table can then be accessed from the application using the linker-generated symbol.

Each `table()` specification you apply to members of a given UNION must contain a unique name. If a `table()` operator is applied to a GROUP, then none of that GROUP's members may be marked with a `table()` specification. The linker detects violations of these rules and reports them as warnings, ignoring each offending use of the `table()` specification. The linker does not generate a copy table for erroneous `table()` operator specifications.

7.15.6 Boot-Time Copy Tables

The linker supports a special copy table name, BINIT (or binit), that you can use to create a boot-time copy table. For example, the link command file for the boot-loaded application described in [Section 7.15.2](#) can be rewritten as follows:

```
SECTIONS
{
    .flashcode: { app_tasks.obj(.text) }
    load = FLASH, run = PMEM,
    table(BINIT)
    ...
}
```

For this example, the linker creates a copy table that can be accessed through a special linker-generated symbol, `__binit__`, which contains the list of all object components that need to be copied from their load location to their run location at boot-time. If a link command file does not contain any uses of `table(BINIT)`, then the `__binit__` symbol is given a value of -1 to indicate that a boot-time copy table does not exist for a particular application.

You can apply the `table(BINIT)` specification to an output section, GROUP, or UNION member. If used in the context of a UNION, only one member of the UNION can be designated with `table(BINIT)`. If applied to a GROUP, then none of that GROUP's members may be marked with `table(BINIT)`. The linker detects violations of these rules and reports them as warnings, ignoring each offending use of the `table(BINIT)` specification.

7.15.7 Using the table() Operator to Manage Object Components

If you have several pieces of code that need to be managed together, then you can apply the same `table()` operator to several different object components. In addition, if you want to manage a particular object component in multiple ways, you can apply more than one `table()` operator to it. Consider the link command file excerpt in [Example 7-18](#):

Example 7-18. Linker Command File to Manage Object Components

```
SECTIONS
{
    UNION
    {
        .first: { a1.obj(.text), b1.obj(.text), c1.obj(.text) }
        load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)

        .second: { a2.obj(.text), b2.obj(.text) }
        load = EMEM, run = PMEM, table(_second_ctbl)
    }

    .extra: load = EMEM, run = PMEM, table(BINIT)
    ...
}
```

In this example, the output sections `.first` and `.extra` are copied from external memory (EMEM) into program memory (PMEM) at boot time while processing the BINIT copy table. After the application has started executing its main thread, it can then manage the contents of the overlay using the two overlay copy tables named: `_first_ctbl` and `_second_ctbl`.

7.15.8 Copy Table Contents

In order to use a copy table that is generated by the linker, you must be aware of the contents of the copy table. This information is included in a new run-time-support library header file, `cpy_tbl.h`, which contains a C source representation of the copy table data structure that is automatically generated by the linker.

[Example 7-19](#) shows the MSP430 copy table header file.

Example 7-19. MSP430 `cpy_tbl.h` File

```

/*****
/* cpy_tbl.h v3.0.0
/* Copyright (c) 2003 Texas Instruments Incorporated
/*
/* Specification of copy table data structures which can be automatically
/* generated by the linker (using the table() operator in the LCF).
/*
/*****

#ifndef _CPY_TBL
#define _CPY_TBL

#ifdef __cplusplus
extern "C" namespace std {
#endif /* __cplusplus */

/*****
/* Copy Record Data Structure
/*****
typedef struct copy_record
{
    unsigned long load_addr;
    unsigned long run_addr;
    unsigned long size;
} COPY_RECORD;

/*****
/* Copy Table Data Structure
/*****
typedef struct copy_table
{
    unsigned short rec_size;
    unsigned short num_recs;
    COPY_RECORD recs[1];
} COPY_TABLE;

/*****
/* Prototype for general purpose copy routine.
/*****
extern void copy_in(COPY_TABLE *tp);

#ifdef __cplusplus
} /* extern "C" namespace std */

#ifndef _CPP_STYLE_HEADER
using std::COPY_RECORD;
using std::COPY_TABLE;
using std::copy_in;
#endif /* _CPP_STYLE_HEADER */
#endif /* __cplusplus */
#endif /* !_CPY_TBL */

```

For each object component that is marked for a copy, the linker creates a COPY_RECORD object for it. Each COPY_RECORD contains at least the following information for the object component:

- The load address
- The run address
- The size

The linker collects all COPY_RECORDs that are associated with the same copy table into a COPY_TABLE object. The COPY_TABLE object contains the size of a given COPY_RECORD, the number of COPY_RECORDs in the table, and the array of COPY_RECORDs in the table. For instance, in the BINIT example in [Section 7.15.6](#), the .first and .extra output sections will each have their own COPY_RECORD entries in the BINIT copy table. The BINIT copy table will then look like this:

```
COPY_TABLE __binit__ = { 12, 2,
    { <load address of .first>,
      <run address of .first>,
      <size of .first> },
    { <load address of .extra>,
      <run address of .extra>,
      <size of .extra> } };
```

7.15.9 General Purpose Copy Routine

The cpy_tbl.h file in [Example 7-19](#) also contains a prototype for a general-purpose copy routine, copy_in(), which is provided as part of the run-time-support library. The copy_in() routine takes a single argument: the address of a linker-generated copy table. The routine then processes the copy table data object and performs the copy of each object component specified in the copy table.

The copy_in() function definition is provided in the cpy_tbl.c run-time-support source file shown in [Example 7-20](#).

Example 7-20. Run-Time-Support cpy_tbl.c File

```

/*****
/* cpy_tbl.c v3.0.0
/* Copyright (c) 2003-2008 Texas Instruments Incorporated
/*
/* General purpose copy routine. Given the address of a linker-generated
/* COPY_TABLE data structure, effect the copy of all object components
/* that are designated for copy via the corresponding LCF table() operator.
/*
/*****
#include <cpy_tbl.h>
#include <string.h>

/*****
/* COPY_IN()
/*****
void copy_in(COPY_TABLE *tp)
{
    unsigned short i;

    for (i = 0; i < tp->num_recs; i++)
    {
        COPY_RECORD crp = tp->recs[i];
#ifdef __LARGE_DATA_MODEL__
        unsigned char *ld_addr = (unsigned char *)crp.load_addr;
        unsigned char *rn_addr = (unsigned char *)crp.run_addr;
#else
        unsigned char *ld_addr = (unsigned char *) (unsigned short)crp.load_addr;
        unsigned char *rn_addr = (unsigned char *) (unsigned short)crp.run_addr;
#endif
        memcpy(rn_addr, ld_addr, crp.size);
    }
}
```

7.15.10 Linker-Generated Copy Table Sections and Symbols

The linker creates and allocates a separate input section for each copy table that it generates. Each copy table symbol is defined with the address value of the input section that contains the corresponding copy table.

The linker generates a unique name for each overlay copy table input section. For example, `table(_first_ctbl)` would place the copy table for the `.first` section into an input section called `.ovly:_first_ctbl`. The linker creates a single input section, `.binit`, to contain the entire boot-time copy table.

[Example 7-21](#) illustrates how you can control the placement of the linker-generated copy table sections using the input section names in the link command file.

Example 7-21. Controlling the Placement of the Linker-Generated Copy Table Sections

```
SECTIONS
{
  UNION
  {
    .first: { a1.obj(.text), b1.obj(.text), c1.obj(.text) }
           load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)

    .second: { a2.obj(.text), b2.obj(.text) }
            load = EMEM, run = PMEM, table(_second_ctbl)
  }

  .extra: load = EMEM, run = PMEM, table(BINIT)

  ...

  .ovly: { } > BMEM
  .binit: { } > BMEM
}
```

For the link command file in [Example 7-21](#), the boot-time copy table is generated into a `.binit` input section, which is collected into the `.binit` output section, which is mapped to an address in the BMEM memory area. The `_first_ctbl` is generated into the `.ovly:_first_ctbl` input section and the `_second_ctbl` is generated into the `.ovly:_second_ctbl` input section. Since the base names of these input sections match the name of the `.ovly` output section, the input sections are collected into the `.ovly` output section, which is then mapped to an address in the BMEM memory area.

If you do not provide explicit placement instructions for the linker-generated copy table sections, they are allocated according to the linker's default placement algorithm.

The linker does not allow other types of input sections to be combined with a copy table input section in the same output section. The linker does not allow a copy table section that was created from a partial link session to be used as input to a succeeding link session.

7.15.11 Splitting Object Components and Overlay Management

In previous versions of the linker, splitting sections that have separate load and run placement instructions was not permitted. This restriction was because there was no effective mechanism for you, the developer, to gain access to the load address or run address of each one of the pieces of the split object component. Therefore, there was no effective way to write a copy routine that could move the split section from its load location to its run location.

However, the linker can access both the load address and run address of every piece of a split object component. Using the `table()` operator, you can tell the linker to generate this information into a copy table. The linker gives each piece of the split object component a `COPY_RECORD` entry in the copy table object.

For example, consider an application which has seven tasks. Tasks 1 through 3 are overlaid with tasks 4 through 7 (using a UNION directive). The load placement of all of the tasks is split among four different memory areas (LMEM1, LMEM2, LMEM3, and LMEM4). The overlay is defined as part of memory area PMEM. You must move each set of tasks into the overlay at run time before any services from the set are used.

You can use table() operators in combination with splitting operators, >>, to create copy tables that have all the information needed to move either group of tasks into the memory overlay as shown in [Example 7-22](#). [Example 7-23](#) illustrates a possible driver for such an application.

Example 7-22. Creating a Copy Table to Access a Split Object Component

```
SECTIONS
{
  UNION
  {
    .task1to3: { *(.task1), *(.task2), *(.task3) }
               load >> LMEM1 | LMEM2 | LMEM4, table(_task13_ctbl)

    GROUP
    {
      .task4: { *(.task4) }
      .task5: { *(.task5) }
      .task6: { *(.task6) }
      .task7: { *(.task7) }

    } load >> LMEM1 | LMEM3 | LMEM4, table(_task47_ctbl)
  } run = PMEM

  ...

  .ovly: > LMEM4
}
```

Example 7-23. Split Object Component Driver

```
#include <copy_tbl.h>

extern far COPY_TABLE task13_ctbl;
extern far COPY_TABLE task47_ctbl;

extern void task1(void);
...
extern void task7(void);

main()
{
  ...
  copy_in(&task13_ctbl);
  task1();
  task2();
  task3();
  ...

  copy_in(&task47_ctbl);
  task4();
  task5();
  task6();
  task7();
  ...
}
```

You must declare a COPY_TABLE object as *far* to allow the overlay copy table section placement to be independent from the other sections containing data objects (such as .bss).

The contents of the .task1to3 section are split in the section's load space and contiguous in its run space. The linker-generated copy table, _task13_ctbl, contains a separate COPY_RECORD for each piece of the split section .task1to3. When the address of _task13_ctbl is passed to copy_in(), each piece of .task1to3 is copied from its load location into the run location.

The contents of the GROUP containing tasks 4 through 7 are also split in load space. The linker performs the GROUP split by applying the split operator to each member of the GROUP in order. The copy table for the GROUP then contains a COPY_RECORD entry for every piece of every member of the GROUP. These pieces are copied into the memory overlay when the _task47_ctbl is processed by copy_in().

The split operator can be applied to an output section, GROUP, or the load placement of a UNION or UNION member. The linker does not permit a split operator to be applied to the run placement of either a UNION or of a UNION member. The linker detects such violations, emits a warning, and ignores the offending split operator usage.

7.16 Partial (Incremental) Linking

An output file that has been linked can be linked again with additional modules. This is known as *partial linking* or *incremental linking*. Partial linking allows you to partition large applications, link each part separately, and then link all the parts together to create the final executable program.

Follow these guidelines for producing a file that you will relink:

- The intermediate files produced by the linker *must* have relocation information. Use the --relocatable option when you link the file the first time. (See [Section 7.4.2.2](#).)
- Intermediate files *must* have symbolic information. By default, the linker retains symbolic information in its output. Do not use the --no_sym_table option if you plan to relink a file, because --no_sym_table strips symbolic information from the output module. (See [Section 7.4.23](#).)
- Intermediate linkers should be concerned only with the formation of output sections and not with allocation. All allocation, binding, and MEMORY directives should be performed in the final linker.
- If the intermediate files have global symbols that have the same name as global symbols in other files and you want them to be treated as static (visible only within the intermediate file), you must link the files with the --make_static option (see [Section 7.4.18](#)).
- If you are linking C code, do not use --ram_model or --rom_model until the final linker. Every time you invoke the linker with the --ram_model or --rom_model option, the linker attempts to create an entry point. (See [Section 7.4.25](#).)

The following example shows how you can use partial linking:

Step 1: Link the file file1.com; use the --relocatable option to retain relocation information in the output file tempout1.out.

```
c1430 --run_linker --relocatable --output_file=tempout1 file1.com
```

file1.com contains:

```
SECTIONS { ss1: { f1.obj f2.obj . . . fn.obj } }
```

Step 2: Link the file file2.com; use the --relocatable option to retain relocation information in the output file tempout2.out.

```
c1430 --run_linker --relocatable --output_file=tempout2 file2.com
```

file2.com contains:

```
SECTIONS { ss2: { g1.obj g2.obj . . . gn.obj } }
```

Step 3: Link tempout1.out and tempout2.out.

```
c1430 --run_linker --map_file=final.map --output_file=final.out tempout1.out  
tempout2.out
```

7.17 Linking C/C++ Code

The C/C++ compiler produces assembly language source code that can be assembled and linked. For example, a C program consisting of modules prog1, prog2, etc., can be assembled and then linked to produce an executable file called prog.out:

```
c1430 --run_linker --rom_model --output_file prog.out prog1.obj prog2.obj ... rts430.lib
```

The `--rom_model` option tells the linker to use special conventions that are defined by the C/C++ environment.

The archive libraries shipped by TI contain C/C++ run-time-support functions.

C, C++, and mixed C and C++ programs can use the same run-time-support library. Run-time-support functions and variables that can be called and referenced from both C and C++ will have the same linkage.

For more information about the MSP430 C/C++ language, including the run-time environment and run-time-support functions, see the *MSP430 Optimizing C/C++ Compiler User's Guide*

7.17.1 Run-Time Initialization

All C/C++ programs must be linked with code to initialize and execute the program, called a *bootstrap* routine, also known as the *boot.obj* object module. The symbol `_c_int00` is defined as the program entry point and is the start of the C boot routine in *boot.obj*; referencing `_c_int00` ensures that *boot.obj* is automatically linked in from the run-time-support library. When a program begins running, it executes *boot.obj* first. The *boot.obj* symbol contains code and data for initializing the run-time environment and performs the following tasks:

- Changes from system mode to user mode
- Sets up the user mode stack
- Processes the run-time *.cinit* initialization table and autoinitializes global variables (when the linker is invoked with the `--rom_model` option)
- Calls `main`

The run-time-support object libraries contain *boot.obj*. You can:

- Use the archiver to extract *boot.obj* from the library and then link the module in directly.
- Include the appropriate run-time-support library as an input file (the linker automatically extracts *boot.obj* when you use the `--ram_model` or `--rom_model` option).

7.17.2 Object Libraries and Run-Time Support

The *MSP430 Optimizing C/C++ Compiler User's Guide* describes additional run-time-support functions that are included in *rts.src*. If your program uses any of these functions, you must link the appropriate run-time-support library with your object files.

You can also create your own object libraries and link them. The linker includes and links only those library members that resolve undefined references.

7.17.3 Setting the Size of the Stack and Heap Sections

The C/C++ language uses two uninitialized sections called *.system* and *.stack* for the memory pool used by the `malloc()` functions and the run-time stacks, respectively. You can set the size of these by using the `--heap_size` or `--stack_size` option and specifying the size of the section as a 4-byte constant immediately after the option. If the options are not used, the default size of the heap is 128 bytes and the default size of the stack is 128 bytes.

See [Section 7.4.13](#) for setting heap sizes and [Section 7.4.30](#) for setting stack sizes.

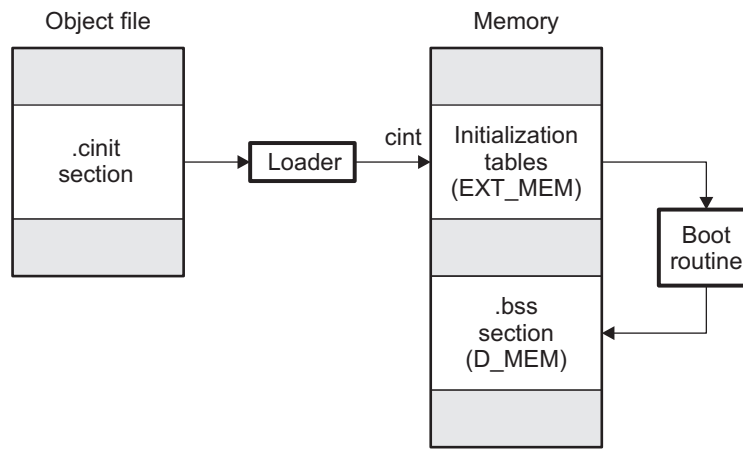
7.17.4 Autoinitialization of Variables at Run Time

Autoinitializing variables at run time is the default method of autoinitialization. To use this method, invoke the linker with the `--rom_model` option.

Using this method, the `.cinit` section is loaded into memory along with all the other initialized sections. The linker defines a special symbol called `cinit` that points to the beginning of the initialization tables in memory. When the program begins running, the C boot routine copies data from the tables (pointed to by `.cinit`) into the specified variables in the `.bss` section. This allows initialization data to be stored in slow external memory and copied to fast external memory each time the program starts.

Figure 7-5 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into slow external memory.

Figure 7-5. Autoinitialization at Run Time



7.17.5 Initialization of Variables at Load Time

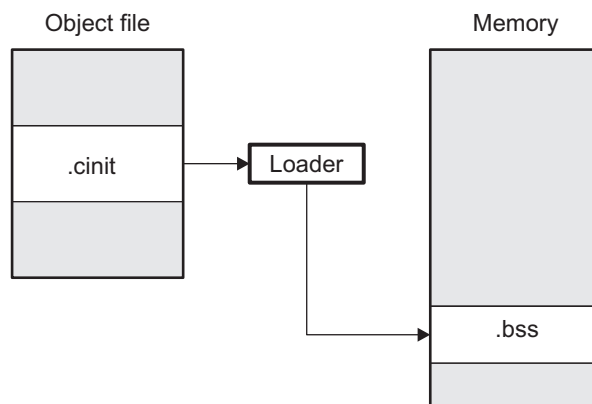
Initialization of variables at load time enhances performance by reducing boot time and by saving the memory used by the initialization tables. To use this method, invoke the linker with the `--ram_model` option.

When you use the `--ram_model` linker option, the linker sets the `STYP_COPY` bit in the `.cinit` section's header. This tells the loader not to load the `.cinit` section into memory. (The `.cinit` section occupies no space in the memory map.) The linker also sets the `cinit` symbol to `-1` (normally, `cinit` points to the beginning of the initialization tables). This indicates to the boot routine that the initialization tables are not present in memory; accordingly, no run-time initialization is performed at boot time.

A loader must be able to perform the following tasks to use initialization at load time:

- Detect the presence of the `.cinit` section in the object file.
- Determine that `STYP_COPY` is set in the `.cinit` section header, so that it knows not to copy the `.cinit` section into memory.
- Understand the format of the initialization tables.

Figure 7-6 illustrates the initialization of variables at load time.

Figure 7-6. Initialization at Load Time


7.17.6 The `--rom_model` and `--ram_model` Linker Options

The following list outlines what happens when you invoke the linker with the `--ram_model` or `--rom_model` option.

- The symbol `_c_int00` is defined as the program entry point. The `_c_int00` symbol is the start of the C boot routine in `boot.obj`; referencing `_c_int00` ensures that `boot.obj` is automatically linked in from the appropriate run-time-support library.
- The `.cinit` output section is padded with a termination record to designate to the boot routine (autoinitialize at run time) or the loader (initialize at load time) when to stop reading the initialization tables.
- When you initialize at load time (`--ram_model` option):
 - The linker sets `cinit` to `-1`. This indicates that the initialization tables are not in memory, so no initialization is performed at run time.
 - The `STYP_COPY` flag (`0010h`) is set in the `.cinit` section header. `STYP_COPY` is the special attribute that tells the loader to perform initialization directly and not to load the `.cinit` section into memory. The linker does not allocate space in memory for the `.cinit` section.
- When you autoinitialize at run time (`--rom_model` option), the linker defines `cinit` as the starting address of the `.cinit` section. The C boot routine uses this symbol as the starting point for autoinitialization.

7.18 Linker Example

This example links three object files named `demo.obj`, `ctrl.obj`, and `tables.obj` and creates a program called `demo.out`. The symbol `SETUP` is the program entry point.

Assume that target memory has the following program memory configuration:

| Address Range | Contents |
|-------------------|----------|
| 0x0200 to 0x0A00 | RAM |
| 0x1100 to 0xFFFF0 | FLASH |
| 0xFFE0 to 0xFFFF | VECTORS |

The output sections are constructed in the following manner:

- Executable code, contained in the `.text` sections of `demo.obj`, `fft.obj`, and `tables.obj`, is linked into program memory ROM.
- Variables, contained in the `var_defs` section of `demo.obj`, are linked into data memory in block `FAST_MEM_2`.
- Tables of coefficients in the `.data` sections of `demo.obj`, `tables.obj`, and `fft.obj` are linked into `FAST_MEM_1`. A hole is created with a length of 100 and a fill value of `0x07A1C`.
- The `xy` section from `demo.obj`, which contains buffers and variables, is linked by default into page 1 of the block `STACK`, since it is not explicitly linked.
- Executable code, contained in the `.text` sections of `demo.obj`, `ctrl.obj`, and `tables.obj`, must be linked into `FLASH`.
- A set of interrupt vectors, contained in the `.intvecs` section of `tables.obj`, must be linked at address `0xFFE0`.
- A table of coefficients, contained in the `.data` section of `tables.obj`, must be linked into `FLASH`. The remainder of block `FLASH` must be initialized to the value `0xFF00`.
- A set of variables, contained in the `.bss` section of `ctrl.obj`, must be linked into `RAM` and preinitialized to `0x0100`.
- Another `.bss` section in `ctrl.obj` must be linked into `RAM`.

[Example 7-24](#) shows the link command file for this example. [Example 7-25](#) shows the map file.

Example 7-24. Linker Command File, demo.cmd

```

/*****
/*                               Specify Linker Options                               */
/*****
--entry_point=SETUP             /* Define the program entry point             */
--output_file=demo.out         /* Name the output file                 */
--map_file=demo.map            /* Create an output map file           */

/*****
/*                               Specify Input Files                               */
/*****
demo.obj
ctrl.obj
tables.obj

/*****
/*                               Specify System Memory Map                       */
/*****

MEMORY
{
    SFR(R)           : origin = 0x0000, length = 0x0010
    PERIPHERALS_8BIT : origin = 0x0010, length = 0x00F0
    PERIPHERALS_16BIT: origin = 0x0100, length = 0x0100
    RAM(RW)          : origin = 0x0200, length = 0x0800
    INFOA            : origin = 0x1080, length = 0x0080
    INFOB            : origin = 0x1000, length = 0x0080
    FLASH            : origin = 0x1100, length = 0xEEE0
    VECTORS(R)       : origin = 0xFFE0, length = 0x001E
    RESET            : origin = 0xFFFE, length = 0x0002
}

/*****
/*                               Specify Output Sections                          */
/*****
SECTIONS
{
    .text      : {} > FLASH           /* Link all .text section into flash */
    .intvecs   : {} > 0xFFE0          /* Link interrupt vectors. at 0xFFE0 */
    .data      :                       /* Link .data sections                */
    {
        tables.obj(.data)
        . = 0x400;
    } = 0xFF00 > FLASH               /* Create hole at end of block        */
    } = 0xFF00 > FLASH               /* Fill and link into FLASH           */

    ctrl_vars :                       /* Create new sections for ctrl variables */
    {
        ctrl.obj(.bss)

    } = 0x0100 > RAM                 /* Fill with 0x0100 and link into RAM */

    .bss      : {} > RAM              /* Link remaining .bss sections into RAM */
}

/*****
/*                               End of Command File                             */
/*****

```

Invoke the linker by entering the following command:

```
cl430 --run_linker demo.cmd
```

This creates the map file shown in [Example 7-25](#) and an output file called demo.out that can be run on a MSP430.

Example 7-25. Output Map File, demo.map

```

OUTPUT FILE NAME:  <demo.out>
ENTRY POINT SYMBOL: "SETUP"  address: 000000d4

MEMORY CONFIGURATION

      name      origin      length      attributes      fill
      -----      -----      -----      -----      -----
      P_MEM      00000000      000001000      RWIX
      D_MEM      00001000      000001000      RWIX
      EEPROM     08000000      000000400      RWIX

SECTION ALLOCATION MAP

      output
      section  page      origin      length      attributes/
      -----  ---      -----      -----      -----
      .text    0      00000020      00000138      ctrl.obj (.text)
                       00000020      000000a0      tables.obj (.text)
                       000000c0      00000000      demo.obj (.text)
                       000000c0      00000098

      .intvecs 0      00000000      00000020      tables.obj (.intvecs)
                       00000000      00000020

      .data    0      08000000      00000400      tables.obj (.data)
                       08000000      00000168      --HOLE-- [fill = ff00ff00]
                       08000168      00000298      ctrl.obj (.data)
                       08000400      00000000      demo.obj (.data)
                       08000400      00000000

      ctrl_var 0      00001000      00000500      ctrl.obj (.bss) [fill = 00000100]
                       00001000      00000500

      .bss     0      00001500      00000100      UNINITIALIZED
                       00001500      00000100      demo.obj (.bss)
                       00001600      00000000      tables.obj (.bss)

GLOBAL SYMBOLS
address  name      address  name
-----  ---      -----  ---
00001500 .bss      00000020 clear
08000000 .data     00000020 .text
00000020 .text     000000b8 set
000000d4 SETUP  000000c0 x42
00000020 clear  000000d4 SETUP
08000400 edata  00000158 etext
00001600 end    00001500 .bss
00000158 etext  00001600 end
000000b8 set    08000000 .data
000000c0 x42    08000400 edata

[10 symbols]

```


Absolute Lister Description

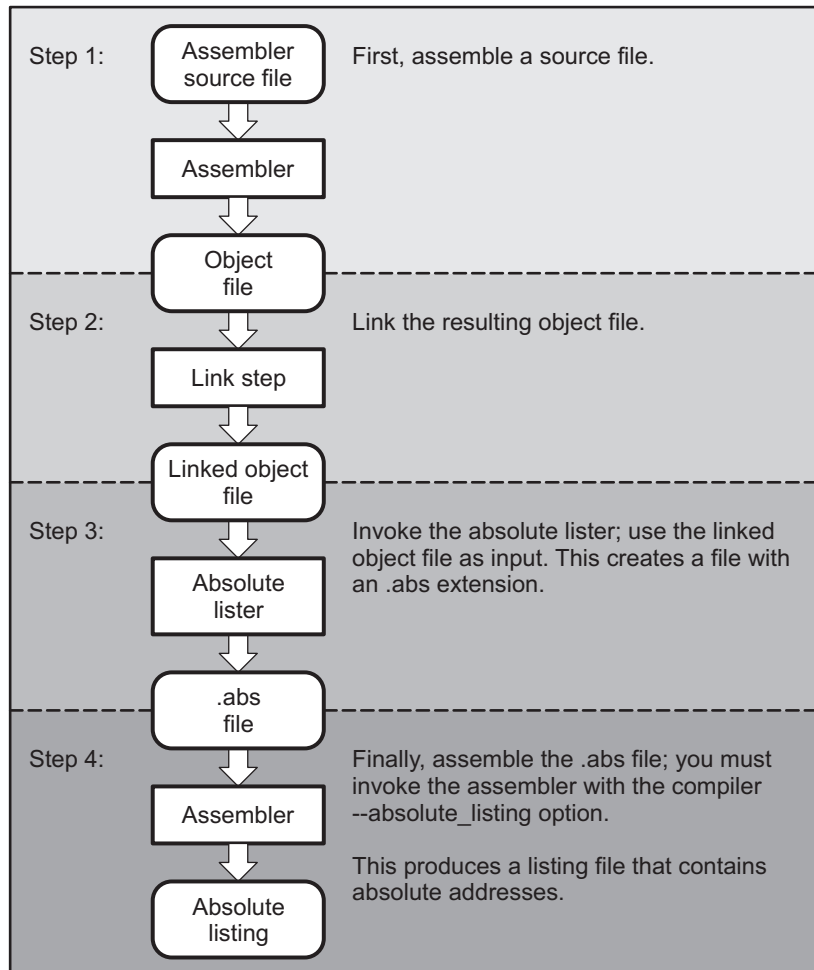
The MSP430™ absolute lister is a debugging tool that accepts linked object files as input and creates .abs files as output. These .abs files can be assembled to produce a listing that shows the absolute addresses of object code. Manually, this could be a tedious process requiring many operations; however, the absolute lister utility performs these operations automatically.

| Topic | Page |
|--|-------------|
| 8.1 Producing an Absolute Listing | 214 |
| 8.2 Invoking the Absolute Lister | 215 |
| 8.3 Absolute Lister Example | 216 |

8.1 Producing an Absolute Listing

Figure 8-1 illustrates the steps required to produce an absolute listing.

Figure 8-1. Absolute Lister Development Flow



8.2 Invoking the Absolute Lister

The syntax for invoking the absolute lister is:

abs430 [-options] *input file*

- abs430** is the command that invokes the absolute lister.
- options* identifies the absolute lister options that you want to use. Options are not case sensitive and can appear anywhere on the command line following the command. Precede each option with a hyphen (-). The absolute lister options are as follows:
- e** enables you to change the default naming conventions for filename extensions on assembly files, C source files, and C header files. The valid options are:
 - ea [.]*asmext* for assembly files (default is .asm)
 - ec [.]*cext* for C source files (default is .c)
 - eh [.]*hext* for C header files (default is .h)
 - ep [.]*pext* for CPP source files (default is cpp)

The . in the extensions and the space between the option and the extension are optional.
 - q** (quiet) suppresses the banner and all progress information.
- input file* names the linked object file. If you do not supply an extension, the absolute lister assumes that the input file has the default extension .out. If you do not supply an input filename when you invoke the absolute lister, the absolute lister prompts you for one.

The absolute lister produces an output file for each file that was linked. These files are named with the input filenames and an extension of .abs. Header files, however, do not generate a corresponding .abs file.

Assemble these files with the --absolute_listing assembler option as follows to create the absolute listing:

cl430 --absolute_listing filename.abs

The -e options affect both the interpretation of filenames on the command line and the names of the output files. They should always precede any filename on the command line.

The -e options are useful when the linked object file was created from C files compiled with the debugging option (--symdebug:dwarf compiler option). When the debugging option is set, the resulting linked object file contains the name of the source files used to build it. In this case, the absolute lister does not generate a corresponding .abs file for the C header files. Also, the .abs file corresponding to a C source file uses the assembly file generated from the C source file rather than the C source file itself.

For example, suppose the C source file hello.csr is compiled with the debugging option set; the debugging option generates the assembly file hello.s. The hello.csr file includes hello.hsr. Assuming the executable file created is called hello.out, the following command generates the proper .abs file:

```
abs430 -ea s -ec csr -eh hsr hello.out
```

An .abs file is not created for hello.hsr (the header file), and hello.abs includes the assembly file hello.s, not the C source file hello.csr.

8.3 Absolute Lister Example

This example uses three source files. The files `module1.asm` and `module2.asm` both include the file `globals.def`.

`module1.asm`

```
.global dflag
.global array
.global offset
.bss dflag,1
.bss array,100
.text
MOV    &array,R11
MOV    &offset,R12
ADD    R12,R11
MOV    @R11,&dflag
```

`module2.asm`

```
.global dflag
.global array
.global offset

.bss offset,1

.text
MOV    &offset,R14
```

`globals.def`

```
.global dflag
.global array
.global offset
```

The following steps create absolute listings for the files `module1.asm` and `module2.asm`:

Step 1. First, assemble `module1.asm` and `module2.asm`:

```
c1430 module1

c1430 module2
```

This creates two object files called `module1.obj` and `module2.obj`.

Step 2. Next, link `module1.obj` and `module2.obj` using the following linker command file, called `bttest.cmd`:

```
--output_file=bttest.out
--map_file=bttest.map
module1.obj
module2.obj
MEMORY
{
    RAM:    origin=0x0200, length=0x0800
    FLASH: origin=0x1100, length=0xEEE0
}
SECTIONS
{
    .bss:  >RAM
    .text: >FLASH
}
```

Invoke the linker:

```
c1430 --run_linker bttest.cmd
```

This command creates an executable object file called `bttest.out`; use this new file as input for the absolute lister.

Step 3. Now, invoke the absolute lister:

```
abs430 bttest.out
```

This command creates two files called module1.abs and module2.abs:

module1.abs:

```
.nolist
.text      .setsym      000001100h
__text__  .setsym      000001100h
etext     .setsym      000001112h
__etext__ .setsym      000001112h
.bss      .setsym      000000200h
__bss__   .setsym      000000200h
end       .setsym      000000266h
__end__   .setsym      000000266h
array     .setsym      000000201h
dflag     .setsym      000000200h
offset    .setsym      000000265h
.setsect  ".text",000001100h
.setsect  ".bss",000000200h
.setsect  ".debug_line",000000000h
.list
.text
.copy     "module1.asm"
```

module2.abs:

```
.nolist
.text      .setsym      000001100h
__text__  .setsym      000001100h
etext     .setsym      000001112h
__etext__ .setsym      000001112h
.bss      .setsym      000000200h
__bss__   .setsym      000000200h
end       .setsym      000000266h
__end__   .setsym      000000266h
array     .setsym      000000201h
dflag     .setsym      000000200h
offset    .setsym      000000265h
.setsect  ".text",00000110eh
.setsect  ".bss",000000265h
.setsect  ".debug_line",00000003ah
.setsect  ".debug_info",00000014fh
.list
.text
.copy     "module2.asm"
```

These files contain the following information that the assembler needs for Step 4:

- They contain .setsym directives, which equate values to global symbols. Both files contain global equates for the symbol *dflag*. The symbol *dflag* was defined in the file *globals.def*, which was included in *module1.asm* and *module2.asm*.
- They contain .setsect directives, which define the absolute addresses for sections.
- They contain .copy directives, which defines the assembly language source file to include.

The .setsym and .setsect directives are useful only for creating absolute listings, not normal assembly.

Step 4. Finally, assemble the .abs files created by the absolute lister (remember that you must use the --absolute_listing option when you invoke the assembler):

```
c1430 --absolute_listing module1.abs
c1430 --absolute_listing module2.abs
```

This command sequence creates two listing files called module1.lst and module2.lst; no object code is produced. These listing files are similar to normal listing files; however, the addresses shown are absolute addresses.

The absolute listing files created are module1.lst (see [Example 8-1](#)) and module2.lst (see [Example 8-2](#)).

Example 8-1. module1.lst

```

module1.abs                                     PAGE      1

    17 001100                                .text
    18                                         .copy      "module1.asm"
A    1                                         .global dflag
A    2                                         .global array
A    3                                         .global offset
A    4 000200                                .bss dflag,1
A    5 000201                                .bss array,100
A    6 001100                                .text
A    7 001100 421B                            MOV &array,R11
    001102 0201!
A    8 001104 421C                            MOV &offset,R12
    001106 0265!
A    9 001108 5C0B                            ADD R12,R11
A   10 00110a 4BA2                            MOV @R11,&dflag
    00110c 0200!

No Assembly Errors, No Assembly Warnings

```

Example 8-2. module2.lst

```

module2.abs                                     PAGE      1

    18 00110e                                .text
    19                                         .copy      "module2.asm"
A    1                                         .global dflag
A    2                                         .global array
A    3                                         .global offset
A    4 000265                                .bss offset,1
A    5 00110e                                .text
A    6 00110e 421E                            MOV &offset,R14
    001110 0265!

No Assembly Errors, No Assembly Warnings

```

Cross-Reference Lister Description

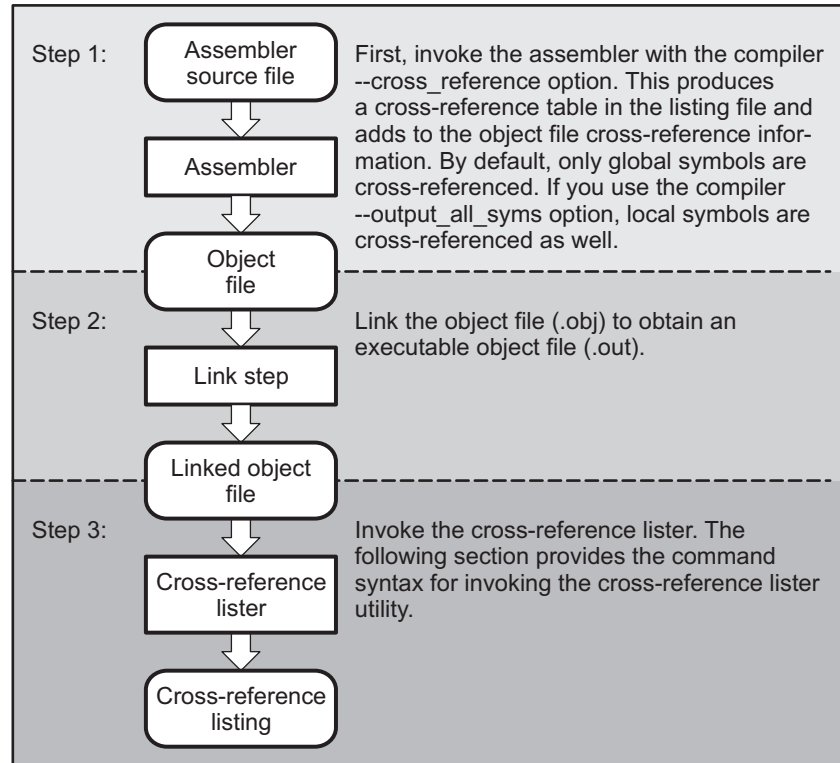
The MSP430™ cross-reference lister is a debugging tool. This utility accepts linked object files as input and produces a cross-reference listing as output. This listing shows symbols, their definitions, and their references in the linked source files.

| Topic | Page |
|--|-------------|
| 9.1 Producing a Cross-Reference Listing | 220 |
| 9.2 Invoking the Cross-Reference Lister | 221 |
| 9.3 Cross-Reference Listing Example | 222 |

9.1 Producing a Cross-Reference Listing

Figure 9-1 illustrates the steps required to produce a cross-reference listing.

Figure 9-1. The Cross-Reference Lister in the MSP430 Software Development Flow



9.2 Invoking the Cross-Reference Lister

To use the cross-reference utility, the file must be assembled with the correct options and then linked into an executable file. Assemble the assembly language files with the `--cross_reference` option. This option creates a cross-reference listing and adds cross-reference information to the object file. By default, the assembler cross-references only global symbols, but if the assembler is invoked with the `--output_all_syms` option, local symbols are also added. Link the object files to obtain an executable file.

To invoke the cross-reference lister, enter the following:

```
xref430 [options] [input filename [output filename]]
```

xref430 is the command that invokes the cross-reference utility.

options identifies the cross-reference lister options you want to use. Options are not case sensitive and can appear anywhere on the command line following the command.

-l (lowercase L) specifies the number of lines per page for the output file. The format of the `-l` option is `-lnum`, where `num` is a decimal constant. For example, `-l30` sets the number of lines per page in the output file to 30. The space between the option and the decimal constant is optional. The default is 60 lines per page.

-q suppresses the banner and all progress information (run quiet).

input filename is a linked object file. If you omit the input filename, the utility prompts for a filename.

output filename is the name of the cross-reference listing file. If you omit the output filename, the default filename is the input filename with an `.xrf` extension.

9.3 Cross-Reference Listing Example

Example 9-1 is an example of cross-reference listing.

Example 9-1. Cross-Reference Listing

```

MSP430 XREF Utility                v3.0.0
2003-2008
File: test.out                    Tue Mar 18 15:35:55 2008        Page:  1

=====

MSP430 XREF Utility                v3.0.0
2003-2008
File: test.out                    Tue Mar 18 15:35:55 2008        Page:  2

=====

Symbol: array

Filename      RTYP    AsmVal    LnkVal    DefLn    RefLn    RefLn    RefLn
-----
module1.asm   EDEF    -0001     0201      5         2         7
=====

MSP430 XREF Utility                v3.0.0
2003-2008
File: test.out                    Tue Mar 18 15:35:55 2008        Page:  3

=====

Symbol: dflag

Filename      RTYP    AsmVal    LnkVal    DefLn    RefLn    RefLn    RefLn
-----
module1.asm   EDEF    -0000     0200      4         1        10
=====

MSP430 XREF Utility                v3.0.0
2003-2008
File: test.out                    Tue Mar 18 15:35:55 2008        Page:  4

=====

Symbol: offset

Filename      RTYP    AsmVal    LnkVal    DefLn    RefLn    RefLn    RefLn
-----
module1.asm   EREF     0000     0265      3         3         8
module2.asm   EDEF    -0000     0265      4         3         6
=====
    
```

The terms defined below appear in the preceding cross-reference listing:

| | |
|-----------------|---|
| Symbol | Name of the symbol listed |
| Filename | Name of the file where the symbol appears |
| RTYP | The symbol's reference type in this file. The possible reference types are: STAT The symbol is defined in this file and is not declared as global. EDEF The symbol is defined in this file and is declared as global. EREF The symbol is not defined in this file but is referenced as global. UNDF The symbol is not defined in this file and is not declared as global. |
| AsmVal | This hexadecimal number is the value assigned to the symbol at assembly time. A value may also be preceded by a character that describes the symbol's attributes. Table 9-1 lists these characters and names. |
| LnkVal | This hexadecimal number is the value assigned to the symbol after linking. |
| DefLn | The statement number where the symbol is defined. |
| RefLn | The line number where the symbol is referenced. If the line number is followed by an asterisk (*), then that reference can modify the contents of the object. A blank in this column indicates that the symbol was never used. |

Table 9-1. Symbol Attributes in Cross-Reference Listing

| Character | Meaning |
|-----------|--|
| ' | Symbol defined in a .text section |
| " | Symbol defined in a .data section |
| + | Symbol defined in a .sect section |
| - | Symbol defined in a .bss or .usect section |

Object File Utilities Descriptions

This chapter describes how to invoke the following miscellaneous utilities:

- The **object file display utility** prints the contents of object files, executable files, and/or archive libraries in both text and XML formats.
- The **disassembler** accepts object files and executable files as input and produces an assembly listing as output. This listing shows assembly instructions, their opcodes, and the section program counter values.
- The **name utility** prints a list of names defined and referenced in an object file, executable files, and/or archive libraries.
- The **strip utility** removes symbol table and debugging information from object and executable files.

| Topic | Page |
|--|------------|
| 10.1 Invoking the Object File Display Utility | 226 |
| 10.2 Invoking the Disassembler | 227 |
| 10.3 Invoking the Name Utility | 228 |
| 10.4 Invoking the Strip Utility | 229 |

10.1 Invoking the Object File Display Utility

The object file display utility, *ofd430*, prints the contents of object files (.obj), executable files (.out), and/or archive libraries (.lib) in both text and XML formats. Hidden symbols are listed as *no name*, while localized symbols are listed like any other local symbol.

To invoke the object file display utility, enter the following:

ofd430 [*options*] *input filename* [*input filename*]

| | |
|-----------------------------------|--|
| ofd430 | is the command that invokes the object file display utility. |
| <i>input filename</i> | names the object file (.obj), executable file (.out), or archive library (.lib) source file. The filename must contain an extension. |
| <i>options</i> | identify the object file display utility options that you want to use. Options are not case sensitive and can appear anywhere on the command line following the command. Precede each option with a hyphen. |
| --dwarf_display=attributes | controls the DWARF display filter settings by specifying a comma-delimited list of <i>attributes</i> . When prefixed with <i>no</i> , an attribute is disabled instead of enabled. Examples: --dwarf_display=nodabbrev,nodline --dwarf_display=all,nodabbrev --dwarf_display=none,dinfo,types The ordering of attributes is important (see --obj_display). The list of available display attributes can be obtained by invoking <i>ofd430 --dwarf_display=help</i> . |
| -g | appends DWARF debug information to program output. |
| -h | displays help |
| -o=filename | sends program output to <i>filename</i> rather than to the screen. |
| --obj_display attributes | controls the object file display filter settings by specifying a comma-delimited list of <i>attributes</i> . When prefixed with <i>no</i> , an attribute is disabled instead of enabled. Examples: --obj_display=rawdata,nostrings --obj_display=all,norawdata --obj_display=none,header The ordering of attributes is important. For instance, in "--obj_display=none,header", <i>ofd430</i> disables all output, then re-enables file header information. If the attributes are specified in the reverse order, (header,none), the file header is enabled, the all output is disabled, including the file header. Thus, nothing is printed to the screen for the given files. The list of available display attributes can be obtained by invoking <i>ofd430 --obj_display=help</i> . |
| -v | prints verbose text output. |
| -x | displays output in XML format. |
| --xml_indent=num | sets the number of spaces to indent nested XML tags. |

If an archive file is given as input to the object file display utility, each object file member of the archive is processed as if it was passed on the command line. The object file members are processed in the order in which they appear in the archive file.

If the object file display utility is invoked without any options, it displays information about the contents of the input files on the console screen.

Object File Display Format

Note: The object file display utility produces data in a text format by default. This data is not intended to be used as input to programs for further processing of the information. XML format should be used for mechanical processing..

10.2 Invoking the Disassembler

The disassembler, *dis430*, examines the output of the assembler or linker. This utility accepts an object file or executable file as input and writes the disassembled object code to standard output or a specified file.

To invoke the disassembler, enter the following:

```
dis430 [options] input filename[.] [output filename]
```

dis430 is the command that invokes the disassembler.

options identifies the name utility options you want to use. Options are not case sensitive and can appear anywhere on the command line following the invocation. Precede each option with a hyphen (-). The name utility options are as follows:

- a** disables the display of the branch destination address along with label names within instructions.
- b** displays data as bytes instead of words.
- c** dumps the object file information.
- d** disables display of data sections.
- h** shows the current help screen.
- i** disassembles .data sections as instructions.
- I** disassembles .text sections as data.
- q** (quiet mode) suppresses the banner and all progress information.
- qq** (super quiet mode) suppresses all headers.
- r** uses raw register IDs (R10, R11, etc.).
- R** shows run-time address if different from load-time address.
- s** suppresses printing of opcode and section program counter in the listing. When you use this option along with **-qq**, the disassembly listing looks like the original assembly source file.
- t** suppresses the display of text sections in the listing.

input filename[.ext] is the name of the input file. If the optional extension is not specified, the file is searched for in this order:

1. *infile*
2. *infile.out*, an executable file
3. *infile.obj*, an object file

output filename is the name of the optional output file to which the disassembly will be written. If an output filename is not specified, the disassembly is written to standard output.

10.3 Invoking the Name Utility

The name utility, *nm430*, prints the list of names defined and referenced in an object (.obj) or an executable file (.out). It also prints the symbol value and an indication of the kind of symbol. Hidden symbols are listed as " ".

To invoke the name utility, enter the following:

nm430 [-options] [input filenames]

| | |
|-----------------------|---|
| nm430 | is the command that invokes the name utility. |
| <i>input filename</i> | is an object file (.obj), executable file (.out), or archive library (.lib). |
| <i>options</i> | identifies the name utility options you want to use. Options are not case sensitive and can appear anywhere on the command line following the invocation. Precede each option with a hyphen (-). The name utility options are as follows: |
| -a | prints all symbols. |
| -c | also prints C_NULL symbols for a COFF object module. |
| -d | also prints debug symbols for a COFF object module. |
| -f | prepends file name to each symbol. |
| -g | prints only global symbols. |
| -h | shows the current help screen. |
| -l | produces a detailed listing of the symbol information. |
| -n | sorts symbols numerically rather than alphabetically. |
| -o file | outputs to the given file. |
| -p | causes the name utility to not sort any symbols. |
| -q | (quiet mode) suppresses the banner and all progress information. |
| -r | sorts symbols in reverse order. |
| -t | also prints tag information symbols for a COFF object module. |
| -u | only prints undefined symbols. |

10.4 Invoking the Strip Utility

The strip utility, *strip430*, removes symbol table and debugging information from object and executable files.

To invoke the strip utility, enter the following:

```
strip430 [-p] input filename [input filename]
```

strip430 is the command that invokes the strip utility.

input filename is an object file (.obj) or an executable file (.out).

options identifies the strip utility options you want to use. Options are not case sensitive and can appear anywhere on the command line following the invocation. Precede each option with a hyphen (-). The strip utility option is as follows:

-o filename writes the stripped output to filename.

-p removes all information not required for execution. This option causes more information to be removed than the default behavior, but the object file is left in a state that cannot be linked. This option should be used only with executable (.out) files.

When the strip utility is invoked without the -o option, the input object files are replaced with the stripped version.

Hex Conversion Utility Description

The MSP430™ assembler and linker create object files which are in binary formats that encourage modular programming and provide powerful and flexible methods for managing code segments and target system memory.

Most EPROM programmers do not accept object files as input. The hex conversion utility converts an object file into one of several standard ASCII hexadecimal formats, suitable for loading into an EPROM programmer. The utility is also useful in other applications requiring hexadecimal conversion of an object file (for example, when using debuggers and loaders).

The hex conversion utility can produce these output file formats:

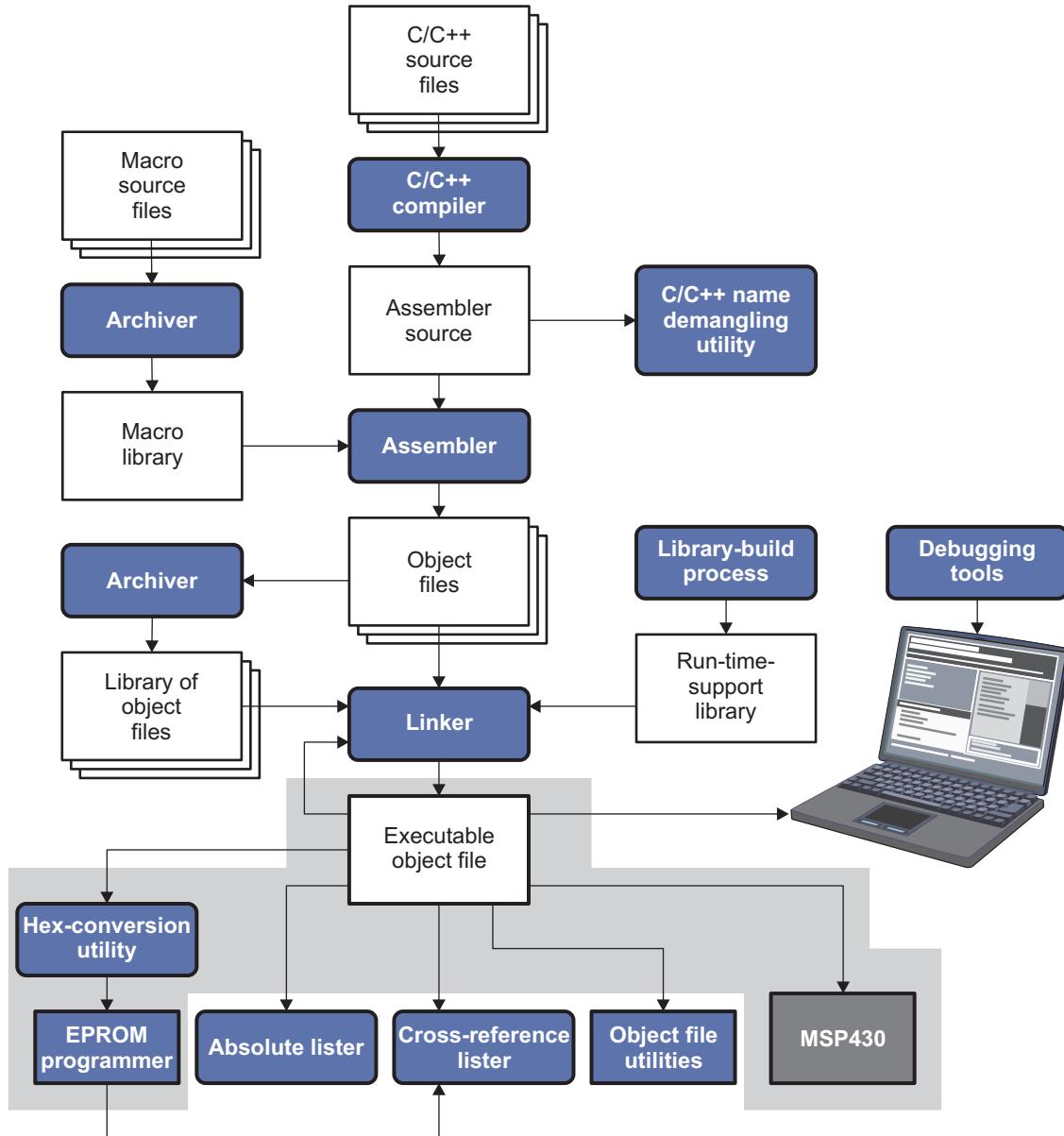
- ASCII-Hex, supporting 16-bit addresses
- Extended Tektronix (Tektronix)
- Intel MCS-86 (Intel)
- Motorola Exorciser (Motorola-S), supporting 16-bit addresses
- Texas Instruments SDSMAC (TI-Tagged), supporting 16-bit addresses
- Texas Instruments TI-TXT format, supporting 16-bit addresses

| Topic | Page |
|---|------------|
| 11.1 The Hex Conversion Utility's Role in the Software Development Flow..... | 232 |
| 11.2 Invoking the Hex Conversion Utility..... | 233 |
| 11.3 Understanding Memory Widths | 236 |
| 11.4 The ROMS Directive..... | 240 |
| 11.5 The SECTIONS Directive | 243 |
| 11.6 Excluding a Specified Section..... | 244 |
| 11.7 Assigning Output Filenames..... | 245 |
| 11.8 Image Mode and the -fill Option | 246 |
| 11.9 Controlling the ROM Device Address..... | 247 |
| 11.10 Description of the Object Formats..... | 248 |

11.1 The Hex Conversion Utility's Role in the Software Development Flow

Figure 11-1 highlights the role of the hex conversion utility in the software development process.

Figure 11-1. The Hex Conversion Utility in the MSP430 Software Development Flow



11.2 Invoking the Hex Conversion Utility

There are two basic methods for invoking the hex conversion utility:

- **Specify the options and filenames on the command line.** The following example converts the file `firmware.out` into TI-Tagged format, producing two output files, `firm.lsb` and `firm.msb`.

```
hex430
```

```
hex430 -t firmware -o firm.lsb -o firm.msb
```

- **Specify the options and filenames in a command file.** You can create a file that stores command line options and filenames for invoking the hex conversion utility. The following example invokes the utility using a command file called `hexutil.cmd`:

```
hex430 hexutil.cmd
```

In addition to regular command line information, you can use the hex conversion utility `ROMS` and `SECTIONS` directives in a command file.

11.2.1 Invoking the Hex Conversion Utility From the Command Line

To invoke the hex conversion utility, enter:

```
hex430 [options] filename
```

hex430 is the command that invokes the hex conversion utility.

options supplies additional information that controls the hex conversion process. You can use options on the command line or in a command file. [Table 11-1](#) lists the basic options.

- All options are preceded by a hyphen and are not case sensitive.
- Several options have an additional parameter that must be separated from the option by at least one space.
- Options with multi-character names must be spelled exactly as shown in this document; no abbreviations are allowed.
- Options are not affected by the order in which they are used. The exception to this rule is the `-q` (quiet) option, which must be used before any other options.

filename names an object file or a command file (for more information, see [Section 11.2.2](#)). If you do not specify a filename, the utility prompts you for one.

Table 11-1. Basic Hex Conversion Utility Options

| General Options | Option | Description | See |
|--|------------------------------------|--|--------------------------------|
| Control the overall operation of the hex conversion utility. | <code>-exclude=section_name</code> | Ignore specified section | Section 11.6 |
| | <code>-h {phrase}</code> | Display the syntax for invoking the compiler and list available options. | Section 11.2.2 |
| | <code>-linkerfill</code> | Include linker fill sections in images | - |
| | <code>-map=filename</code> | Generate a map file | Section 11.4.2 |
| | <code>-o=filename</code> | Specify an output filename | Section 11.7 |
| | <code>-olength=value</code> | Specify maximum number of data items per line of output | - |
| | <code>-quiet, -q</code> | Run quietly (when used, it must appear <i>before</i> other options) | Section 11.2.2 |

Table 11-1. Basic Hex Conversion Utility Options (continued)

| Image Options | Option | Description | See |
|---|-------------------------|---|---------------------------------|
| Create a continuous image of a range of target memory | -fill= <i>value</i> | Fill holes with <i>value</i> | Section 11.8.2 |
| | -image | Specify image mode | Section 11.8.1 |
| | -zero | Reset the address origin to 0 in image mode | Section 11.8.3 |
| Memory Options | Option | Description | See |
| Configure the memory widths for your output files | -memwidth= <i>value</i> | Define the system memory word width (default 16 bits) | Section 11.3.2 |
| | --order=LS | Output file is in little-endian format | Section 11.3.4 |
| | --order=MS | Output file is in big-endian format | Section 11.3.4 |
| | -romwidth= <i>value</i> | Specify the ROM device width (default depends on format used) | Section 11.3.3 |
| Output Options | Option | Description | See |
| Specify the output format | -a | Select ASCII-Hex | |
| | -i | Select Intel | Section 11.10.1 |
| | -m1 | Select Motorola-S1 | Section 11.10.3 |
| | -m2 | Select Motorola-S2 | Section 11.10.3 |
| | -m3 | Select Motorola-S3 (default -m option) | Section 11.10.3 |
| | -t | Select TI-Tagged | Section 11.10.4 |
| | --ti_txt | Select TI-Txt | Section 11.10.5 |
| | -x | Select Tektronix (default format) | Section 11.10.6 |

11.2.2 Invoking the Hex Conversion Utility With a Command File

A command file is useful if you plan to invoke the utility more than once with the same input files and options. It is also useful if you want to use the ROMS and SECTIONS hex conversion utility directives to customize the conversion process.

Command files are ASCII files that contain one or more of the following:

- **Options and filenames.** These are specified in a command file in exactly the same manner as on the command line.
- **ROMS directive.** The ROMS directive defines the physical memory configuration of your system as a list of address-range parameters. (See [Section 11.4.](#))
- **SECTIONS directive.** The hex conversion utility SECTIONS directive specifies which sections from the object file are selected. (See [Section 11.5.](#))
- **Comments.** You can add comments to your command file by using the `/*` and `*/` delimiters. For example:

```
/* This is a comment. */
```

To invoke the utility and use the options you defined in a command file, enter:

```
hex430 command_filename
```

You can also specify other options and files on the command line. For example, you could invoke the utility by using both a command file and command line options:

```
hex430 firmware.cmd -map firmware.mxp
```

The order in which these options and filenames appear is not important. The utility reads all input from the command line and all information from the command file before starting the conversion process. However, if you are using the `-q` option, *it must appear as the first option on the command line or in a command file.*

The **-h option** displays the syntax for invoking the compiler and lists available options. If the **-h** option is followed by another option or phrase, detailed information about the option or phrase is displayed. For example, to see information about options associated with generating a boot table use **-h boot**.

The **-q option** suppresses the hex conversion utility's normal banner and progress information.

- Assume that a command file named `firmware.cmd` contains these lines:

```
firmware.out /* input file */
-t          /* TI-Tagged */
-o firm.lsb /* output file */
-o firm.msb /* output file */
```

You can invoke the hex conversion utility by entering:

```
hex430 firmware.cmd
```

- This example shows how to convert a file called `appl.out` into eight hex files in Intel format. Each output file is one byte wide and 4K bytes long.

```
appl.out /* input file */
-I       /* Intel format */
-map appl.mxp /* map file */
```

```
ROMS
{
  ROW1: origin=0x00000000 len=0x4000 romwidth=8
        files={ appl.u0 appl.u1 appl.u2 appl.u3 }
  ROW2: origin=0x00004000 len=0x4000 romwidth=8
        files={ appl.u4 appl.u5 appl.u6 appl.u7 }
}

SECTIONS
{
  .text, .data, .cinit, .sect1, .vectors, .const:
}
```

11.3 Understanding Memory Widths

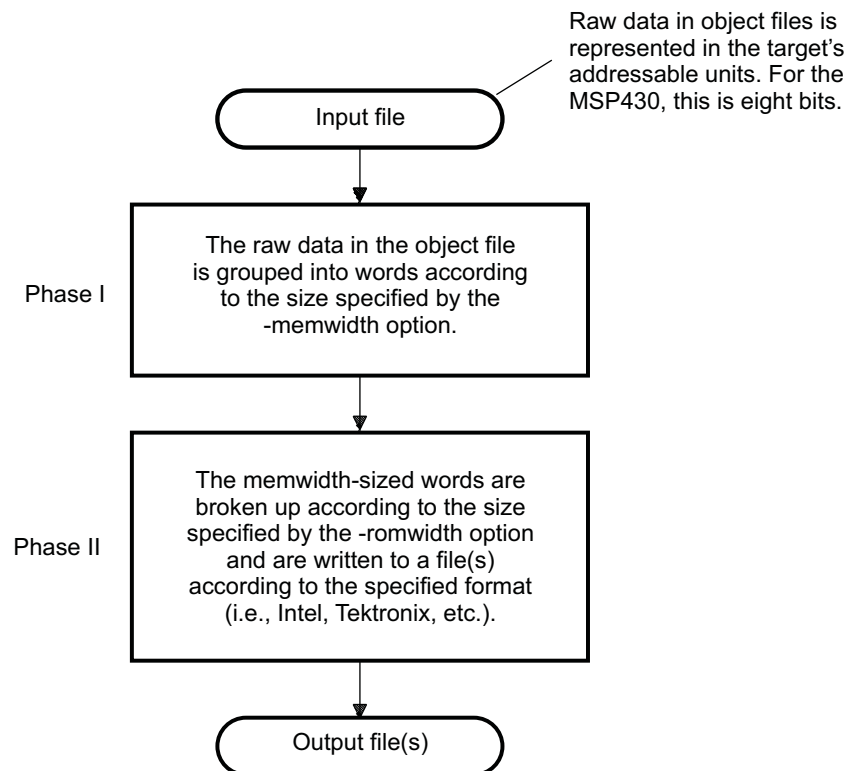
The hex conversion utility makes your memory architecture more flexible by allowing you to specify memory and ROM widths. To use the hex conversion utility, you must understand how the utility treats word widths. Three widths are important in the conversion process:

- Target width
- Memory width
- ROM width

The terms target word, memory word, and ROM word refer to a word of such a width.

Figure 11-2 illustrates the separate and distinct phases of the hex conversion utility's process flow.

Figure 11-2. Hex Conversion Utility Process Flow



11.3.1 Target Width

Target width is the unit size (in bits) of the target processor's word. The unit size corresponds to the data bus size on the target processor. The width is fixed for each target and cannot be changed. The MSP430 targets have a width of 16 bits.

11.3.2 Specifying the Memory Width

Memory width is the physical width (in bits) of the memory system. Usually, the memory system is physically the same width as the target processor width: a 16-bit processor has a 16-bit memory architecture. However, some applications require target words to be broken into multiple, consecutive, and narrower memory words.

By default, the hex conversion utility sets memory width to the target width (in this case, 16 bits).

You can change the memory width (except for TI-TXT format) by:

- Using the **-memwidth** option. This changes the memory width value for the entire file.
- Setting the **memwidth** parameter of the ROMS directive. This changes the memory width value for the address range specified in the ROMS directive and overrides the -memwidth option for that range. See [Section 11.4](#).

For both methods, use a value that is a power of 2 greater than or equal to 8.

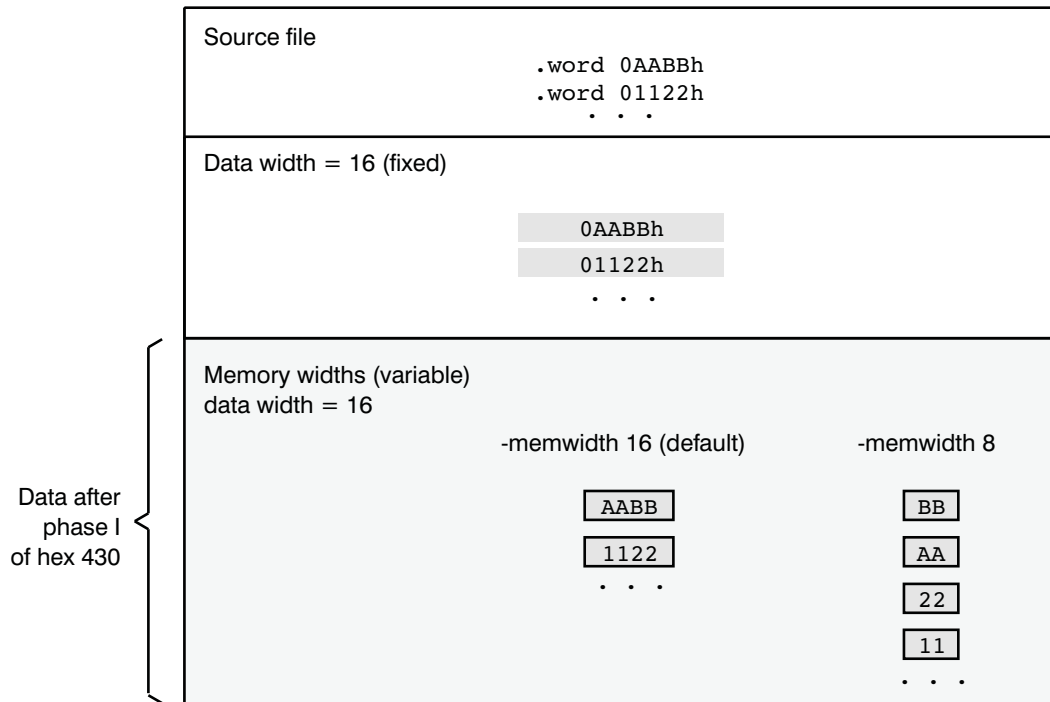
You should change the memory width default value of 16 only when you need to break single target words into consecutive, narrower memory words.

TI-TXT Format is 8 Bits Wide

Note: You cannot change the memory width of the TI-TXT format. The TI-TXT hex format supports an 8-bit memory width only.

Figure 11-3 demonstrates how the memory width is related to object file data.

Figure 11-3. Object File Data and Memory Widths



11.3.3 Partitioning Data Into Output Files

ROM width specifies the physical width (in bits) of each ROM device and corresponding output file (usually one byte or eight bits). The ROM width determines how the hex conversion utility partitions the data into output files. After the object file data is mapped to the memory words, the memory words are broken into one or more output files. The number of output files is determined by the following formulas:

- If memory width \geq ROM width:
number of files = memory width \div ROM width
- If memory width $<$ ROM width:
number of files = 1

For example, for a memory width of 16, you could specify a ROM width value of 16 and get a single output file containing 16-bit words. Or you can use a ROM width value of 8 to get two files, each containing 8 bits of each word.

The default ROM width that the hex conversion utility uses depends on the output format:

- All hex formats except TI-Tagged are configured as lists of 8-bit bytes; the default ROM width for these formats is 8 bits.
- TI-Tagged is a 16-bit format; the default ROM width for TI-Tagged is 16 bits.

The TI-Tagged Format is 16 Bits Wide

Note: You cannot change the ROM width of the TI-Tagged format. The TI-Tagged format supports a 16-bit ROM width only.

TI-TXT Format is 8 Bits Wide

Note: You cannot change the ROM width of the TI-TXT format. The TI-TXT hex format supports only an 8-bit ROM width.

You can change ROM width (except for TI-Tagged and TI-TXT formats) by:

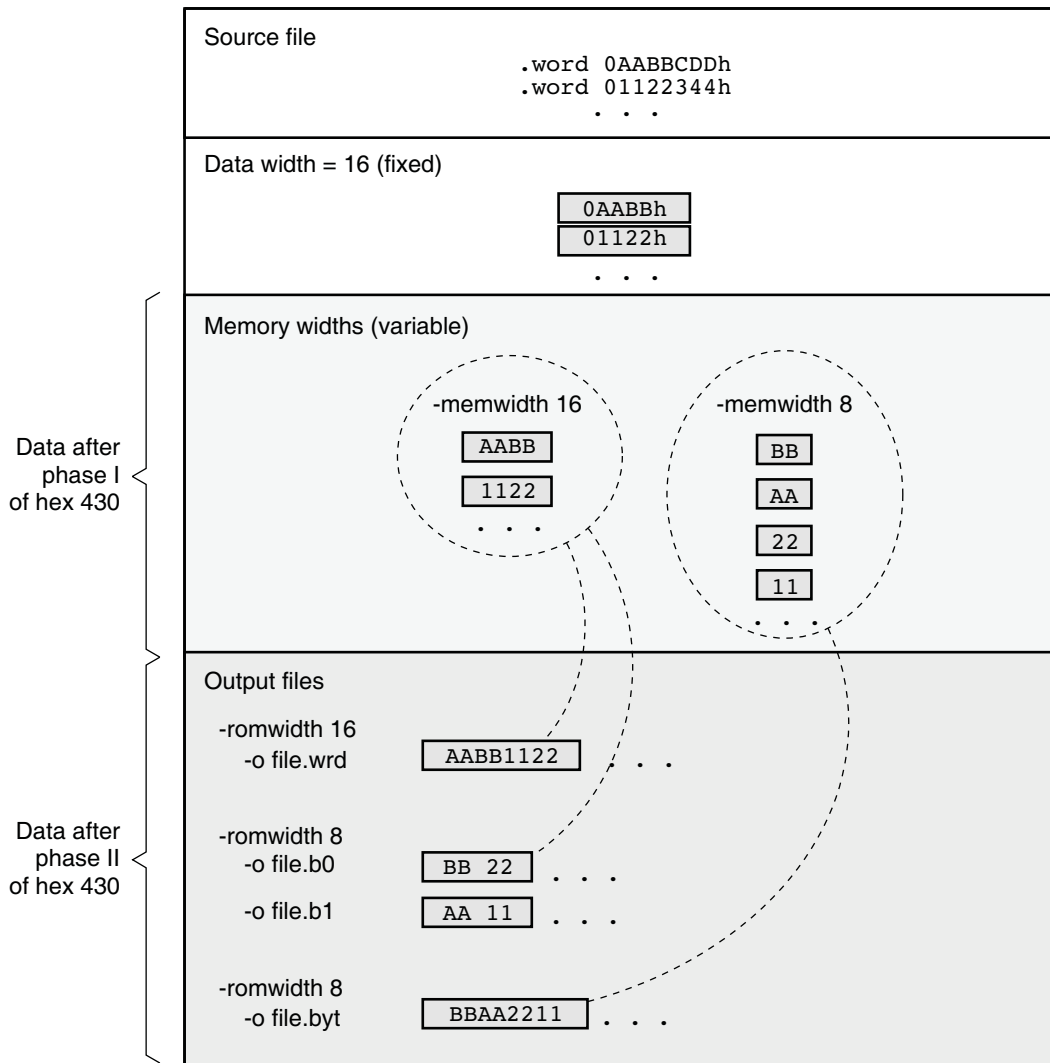
- Using the **-romwidth** option. This option changes the ROM width value for the entire object file.
- Setting the **romwidth** parameter of the ROMS directive. This parameter changes the ROM width value for a specific ROM address range and overrides the -romwidth option for that range. See [Section 11.4](#).

For both methods, use a value that is a power of 2 greater than or equal to 8.

If you select a ROM width that is wider than the natural size of the output format (16 bits for TI-Tagged or 8 bits for all others), the utility simply writes multibyte fields into the file.

[Figure 11-4](#) illustrates how the object file data, memory, and ROM widths are related to one another.

Figure 11-4. Data, Memory, and ROM Widths



11.3.4 Specifying Word Order for Output Words

There are two ways to split a wide word into consecutive memory locations in the same hex conversion utility output file:

- **--order=MS** specifies **big-endian** ordering, in which the most significant part of the wide word occupies the first of the consecutive locations.
- **--order=LS** specifies **little-endian** ordering, in which the least significant part of the wide word occupies the first of the consecutive locations.

By default, the utility uses little-endian format. Unless your boot loader program expects big-endian format, avoid using `--order=MS`.

11.4 The ROMS Directive

The ROMS directive specifies the physical memory configuration of your system as a list of address-range parameters.

Each address range produces one set of files containing the hex conversion utility output data that corresponds to that address range. Each file can be used to program one single ROM device.

The ROMS directive is similar to the MEMORY directive of the MSP430 linker: both define the memory map of the target address space. Each line entry in the ROMS directive defines a specific address range. The general syntax is:

```

ROMS
{
    romname :    [origin=value,] [length=value,] [romwidth=value,]
                 [memwidth=value,] [fill=value]
                 [files={filename1, filename2, ...}]
    romname :    [origin=value,] [length=value,] [romwidth=value,]
                 [memwidth=value,] [fill=value]
                 [files={filename1, filename2, ...}]
    ...
}
    
```

ROMS begins the directive definition.

romname identifies a memory range. The name of the memory range can be one to eight characters in length. The name has no significance to the program; it simply identifies the range. (Duplicate memory range names are allowed.)

origin specifies the starting address of a memory range. It can be entered as origin, org, or o. The associated value must be a decimal, octal, or hexadecimal constant. If you omit the origin value, the origin defaults to 0. The following table summarizes the notation you can use to specify a decimal, octal, or hexadecimal constant:

| Constant | Notation | Example |
|-------------|-----------------------|--------------|
| Hexadecimal | 0x prefix or h suffix | 0x77 or 077h |
| Octal | 0 prefix | 077 |
| Decimal | No prefix or suffix | 77 |

length specifies the length of a memory range as the physical length of the ROM device. It can be entered as length, len, or l. The value must be a decimal, octal, or hexadecimal constant. If you omit the length value, it defaults to the length of the entire address space.

romwidth specifies the physical ROM width of the range in bits (see [Section 11.3.3](#)). Any value you specify here overrides the -romwidth option. The value must be a decimal, octal, or hexadecimal constant that is a power of 2 greater than or equal to 8.

| | |
|-----------------|---|
| memwidth | specifies the memory width of the range in bits (see Section 11.3.2). Any value you specify here overrides the <code>-memwidth</code> option. The value must be a decimal, octal, or hexadecimal constant that is a power of 2 greater than or equal to 8. <i>When using the <code>memwidth</code> parameter, you must also specify the <code>paddr</code> parameter for each section in the <code>SECTIONS</code> directive.</i> (See Section 11.5 .) |
| fill | specifies a fill value to use for the range. In image mode, the hex conversion utility uses this value to fill any holes between sections in a range. A hole is an area between the input sections that comprises an output section that contains no actual code or data. The fill value must be a decimal, octal, or hexadecimal constant with a width equal to the target width. Any value you specify here overrides the <code>-fill</code> option. When using <code>fill</code> , you must also use the <code>-image</code> command line option. (See Section 11.8.2 .) |
| files | identifies the names of the output files that correspond to this range. Enclose the list of names in curly braces and order them from <i>least significant</i> to <i>most significant</i> output file, where the bits of the memory word are numbered from right to left. The number of file names must equal the number of output files that the range generates. To calculate the number of output files, see Section 11.3.3 . The utility warns you if you list too many or too few filenames. |

Unless you are using the `-image` option, all of the parameters that define a range are optional; the commas and equal signs are also optional. A range with no origin or length defines the entire address space. In image mode, an origin and length are required for all ranges.

Ranges must not overlap and must be listed in order of ascending address.

11.4.1 When to Use the ROMS Directive

If you do not use a ROMS directive, the utility defines a single default range that includes the entire address space. This is equivalent to a ROMS directive with a single range without origin or length.

Use the ROMS directive when you want to:

- **Program large amounts of data into fixed-size ROMs.** When you specify memory ranges corresponding to the length of your ROMs, the utility automatically breaks the output into blocks that fit into the ROMs.
- **Restrict output to certain segments.** You can also use the ROMS directive to restrict the conversion to a certain segment or segments of the target address space. The utility does not convert the data that falls outside of the ranges defined by the ROMS directive. Sections can span range boundaries; the utility splits them at the boundary into multiple ranges. If a section falls completely outside any of the ranges you define, the utility does not convert that section and issues no messages or warnings. Thus, you can exclude sections without listing them by name with the `SECTIONS` directive. However, if a section falls partially in a range and partially in unconfigured memory, the utility issues a warning and converts only the part within the range.
- **Use image mode.** When you use the `-image` option, you must use a ROMS directive. Each range is filled completely so that each output file in a range contains data for the whole range. Holes before, between, or after sections are filled with the fill value from the ROMS directive, with the value specified with the `-fill` option, or with the default value of 0.

11.4.2 An Example of the ROMS Directive

The ROMS directive in [Example 11-1](#) shows how 16K bytes of 16-bit memory could be partitioned for two 8K-byte 8-bit EPROMs. [Figure 11-5](#) illustrates the input and output files.

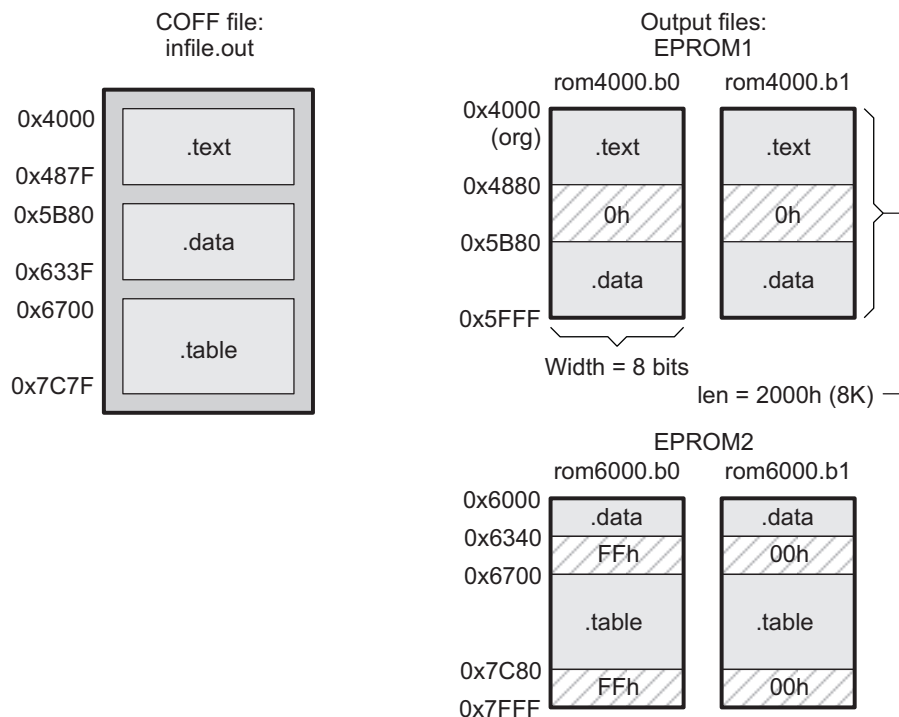
Example 11-1. A ROMS Directive Example

```
infile.out
-image
-memwidth 16

ROMS
{
  EPROM1: org = 0x4000, len = 0x2000, romwidth = 8
         files = { rom4000.b0, rom4000.b1}

  EPROM2: org = 0x6000, len = 0x2000, romwidth = 8,
         fill = 0xFF00,
         files = { rom6000.b0, rom6000.b1}
}
```

Figure 11-5. The infile.out File Partitioned Into Four Output Files



The map file (specified with the -map option) is advantageous when you use the ROMS directive with multiple ranges. The map file shows each range, its parameters, names of associated output files, and a list of contents (section names and fill values) broken down by address. [Example 11-2](#) is a segment of the map file resulting from the example in [Example 11-1](#).

Example 11-2. Map File Output From Example 11-1 Showing Memory Ranges

```

-----
4000..5fff Page=0 Width=8 "EPROM1"
-----
OUTPUT FILES:  rom4000.b0  [b0..b7]
                rom4000.b1  [b8..b15]
CONTENTS: 4000..487f  .text
          4880..5b7f  FILL = 0000
          5b80..5fff  .data
-----
6000..7fff Page=0 Width=8 "EPROM2"
-----
OUTPUT FILES:  rom6000.b0  [b0..b7]
                rom6000.b1  [b8..b15]
CONTENTS: 6000..633f  .data
          6340..66ff  FILL = ff00
          6700..7c7f  .table
          7c80..7fff  FILL = ff00

```

EPROM1 defines the address range from 0x4000 through 0x5FFF with the following sections:

| This section ... | Has this range ... |
|------------------|-----------------------|
| .text | 0x4000 through 0x487F |
| .data | 0x5B80 through 0x5FFF |

The rest of the range is filled with 0h (the default fill value), converted into two output files:

- rom4000.b0 contains bits 0 through 7
- rom4000.b1 contains bits 8 through 15

EPROM2 defines the address range from 0x6000 through 0x7FFF with the following sections:

| This section ... | Has this range ... |
|------------------|-----------------------|
| .data | 0x6000 through 0x633F |
| .table | 0x6700 through 0x7C7F |

The rest of the range is filled with 0xFF00 (from the specified fill value). The data from this range is converted into two output files:

- rom6000.b0 contains bits 0 through 7
- rom6000.b1 contains bits 8 through 15

11.5 The SECTIONS Directive

You can convert specific sections of the object file by name with the hex conversion utility SECTIONS directive. You can also specify those sections that you want to locate in ROM at a different address than the *load* address specified in the linker command file. If you:

- Use a SECTIONS directive, the utility converts only the sections that you list in the directive and ignores all other sections in the object file.
- Do not use a SECTIONS directive, the utility converts all initialized sections that fall within the configured memory.

Uninitialized sections are *never* converted, whether or not you specify them in a SECTIONS directive.

Sections Generated by the C/C++ Compiler

Note: The MSP430 C/C++ compiler automatically generates these sections:

- **Initialized sections:** .text, .const, and .cinit
- **Uninitialized sections:** .bss, .stack, and .systemem

Use the SECTIONS directive in a command file. (See [Section 11.2.2.](#)) The general syntax for the SECTIONS directive is:

SECTIONS

```
{
  sname[:] [paddr=value][,]
  sname[:] [paddr=value][,]
  ...
}
```

SECTIONS begins the directive definition.

sname identifies a section in the input file. If you specify a section that does not exist, the utility issues a warning and ignores the name.

paddr=value specifies the physical ROM address at which this section should be located. This value overrides the section load address given by the linker. This value must be a decimal, octal, or hexadecimal constant. *If your file contains multiple sections, and if one section uses a paddr parameter, then all sections must use a paddr parameter.*

For more similarity with the linker's SECTIONS directive, you can use colons after the section names. For example, the data in your application (section partB) must be loaded on the EPROM at address 0x0. Use the paddr option with the SECTIONS directive to specify this:

```
SECTIONS
{
  partB: paddr = 0x0
}
```

The commas separating section names are optional. For example, the COFF file contains six initialized sections: .text, .data, .const, .vectors, .coeff, and .tables. Suppose you want only .text and .data to be converted. Use a SECTIONS directive to specify this:

```
SECTIONS { .text, .data }
```

11.6 Excluding a Specified Section

The `-exclude section_name` option can be used to inform the hex utility to ignore the specified section. If a SECTIONS directive is used, it overrides the `-exclude` option.

For example, if a SECTIONS directive containing the section name `mysect` is used and an `-exclude mysect` is specified, the SECTIONS directive takes precedence and `mysect` is not excluded.

The `-exclude` option has a limited wildcard capability. The `*` character can be placed at the beginning or end of the name specifier to indicate a suffix or prefix, respectively. For example, `-exclude sect*` disqualifies all sections that begin with the characters `sect`.

If you specify the `-exclude` option on the command line with the `*` wildcard, enter quotes around the section name and wildcard. For example, `-exclude"sect*"`. Using quotes prevents the `*` from being interpreted by the hex conversion utility. If `-exclude` is in a command file, then the quotes should not be specified.

11.7 Assigning Output Filenames

When the hex conversion utility translates your object file into a data format, it partitions the data into one or more output files. When multiple files are formed by splitting memory words into ROM words, *filenames are always assigned in order from least to most significant*, where bits in the memory words are numbered from right to left. This is true, regardless of target or endian ordering.

The hex conversion utility follows this sequence when assigning output filenames:

1. **It looks for the ROMS directive.** If a file is associated with a range in the ROMS directive and you have included a list of files (files = { . . . }) on that range, the utility takes the filename from the list.

For example, assume that the target data is 16-bit words being converted to two files, each eight bits wide. To name the output files using the ROMS directive, you could specify:

```
ROMS
{
  RANGE1: romwidth=8, files={ xyz.b0 xyz.b1 }
}
```

The utility creates the output files by writing the least significant bits to xyz.b0 and the most significant bits to xyz.b1.

2. **It looks for the -o options.** You can specify names for the output files by using the -o option. If no filenames are listed in the ROMS directive and you use -o options, the utility takes the filename from the list of -o options. The following line has the same effect as the example above using the ROMS directive:

```
-o xyz.b0 -o xyz.b1
```

If both the ROMS directive and -o options are used together, the ROMS directive overrides the -o options.

3. **It assigns a default filename.** If you specify no filenames or fewer names than output files, the utility assigns a default filename. A default filename consists of the base name from the input file plus a 2- to 3-character extension. The extension has three parts:

- a. A format character, based on the output format (see [Section 11.10](#)):

| | |
|----------|----------------|
| a | for ASCII-Hex |
| i | for Intel |
| m | for Motorola-S |
| t | for TI-Tagged |
| x | for Tektronix |

- b. The range number in the ROMS directive. Ranges are numbered starting with 0. If there is no ROMS directive, or only one range, the utility omits this character.

- c. The file number in the set of files for the range, starting with 0 for the least significant file.

For example, assume a.out is for a 16-bit target processor and you are creating Intel format output. With no output filenames specified, the utility produces two output files named a.i0, a.i1, a.i2, a.i3.

If you include the following ROMS directive when you invoke the hex conversion utility, you would have four output files:

```
ROMS
{
  range1: o = 0x1000 l = 0x1000
  range2: o = 0x2000 l = 0x1000
}
```

| These output files ... | Contain data in these locations ... |
|------------------------|-------------------------------------|
| a.i00 and a.i01 | 0x1000 through 0x1FFF |
| a.i10 and a.i11 | 0x2000 through 0x2FFF |

11.8 Image Mode and the -fill Option

This section points out the advantages of operating in image mode and describes how to produce output files with a precise, continuous image of a target memory range.

11.8.1 Generating a Memory Image

With the -image option, the utility generates a memory image by completely filling all of the mapped ranges specified in the ROMS directive.

An object file consists of blocks of memory (sections) with assigned memory locations. Typically, all sections are not adjacent: there are holes between sections in the address space for which there is no data. When such a file is converted *without* the use of image mode, the hex conversion utility bridges these holes by using the address records in the output file to skip ahead to the start of the next section. In other words, there may be discontinuities in the output file addresses. Some EPROM programmers do not support address discontinuities.

In image mode, there are no discontinuities. Each output file contains a continuous stream of data that corresponds exactly to an address range in target memory. Any holes before, between, or after sections are filled with a fill value that you supply.

An output file converted by using image mode still has address records, because many of the hexadecimal formats require an address on each line. However, in image mode, these addresses are always contiguous.

Defining the Ranges of Target Memory

Note: If you use image mode, you must also use a ROMS directive. In image mode, each output file corresponds directly to a range of target memory. You must define the ranges. If you do not supply the ranges of target memory, the utility tries to build a memory image of the entire target processor address space. This is potentially a huge amount of output data. To prevent this situation, the utility requires you to explicitly restrict the address space with the ROMS directive.

11.8.2 Specifying a Fill Value

The -fill option specifies a value for filling the holes between sections. The fill value must be specified as an integer constant following the -fill option. The width of the constant is assumed to be that of a word on the target processor. For example, specifying -fill 0x0FF. The constant value is not sign extended.

The hex conversion utility uses a default fill value of 0 if you do not specify a value with the fill option. *The -fill option is valid only when you use -image; otherwise, it is ignored.*

11.8.3 Steps to Follow in Using Image Mode

- Step 1:** Define the ranges of target memory with a ROMS directive. See [Section 11.4](#).
- Step 2:** Invoke the hex conversion utility with the -image option. You can optionally use the -zero option to reset the address origin to 0 for each output file. If you do not specify a fill value with the ROMS directive and you want a value other than the default of 0, use the -fill option.

11.9 Controlling the ROM Device Address

The hex conversion utility output address field corresponds to the ROM device address. The EPROM programmer burns the data into the location specified by the hex conversion utility output file address field. The hex conversion utility offers some mechanisms to control the starting address in ROM of each section. However, many EPROM programmers offer direct control of the location in ROM in which the data is burned.

The address field of the hex-conversion utility output file is controlled by the following items, which are listed from low to high priority:

1. **The linker command file.** By default, the address field of the hex conversion utility output file is the load address (as given in the linker command file).
2. **The `paddr` parameter of the `SECTIONS` directive.** When the `paddr` parameter is specified for a section, the hex conversion utility bypasses the section load address and places the section in the address specified by `paddr`.
3. **The `-zero` option.** When you use the `-zero` option, the utility resets the address origin to 0 for each output file. Since each file starts at 0 and counts upward, any address records represent offsets from the beginning of the file (the address within the ROM) rather than actual target addresses of the data. You must use the `-zero` option in conjunction with the `-image` option to force the starting address in each output file to be zero. If you specify the `-zero` option without the `-image` option, the utility issues a warning and ignores the `-zero` option.

11.10 Description of the Object Formats

The hex conversion utility has options that identify each format. [Table 11-2](#) specifies the format options. They are described in the following sections.

- You need to use only one of these options on the command line. If you use more than one option, the last one you list overrides the others.
- The default format is Tektronix (-x option).

Table 11-2. Options for Specifying Hex Conversion Formats

| Option | Format | Address Bits | Default Width |
|----------|------------|--------------|---------------|
| -a | ASCII-Hex | 16 | 8 |
| -i | Intel | 32 | 8 |
| -m | Motorola-S | 32 | 8 |
| -t | TI-Tagged | 16 | 16 |
| --ti_txt | TI_TXT | 8 | 8 |
| -x | Tektronix | 32 | 8 |

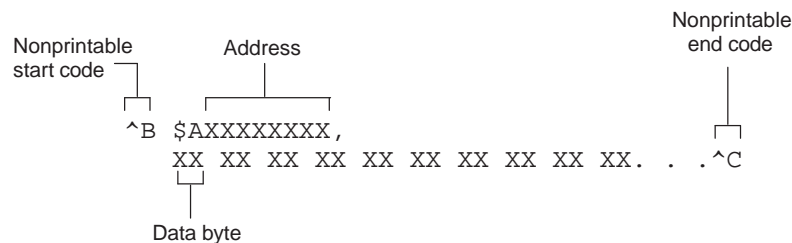
Address bits determine how many bits of the address information the format supports. Formats with 16-bit addresses support addresses up to 64K only. The utility truncates target addresses to fit in the number of available bits.

The **default width** determines the default output width of the format. You can change the default width by using the -romwidth option or by using the romwidth parameter in the ROMS directive. You cannot change the default width of the TI-Tagged format, which supports a 16-bit width only.

11.10.1 ASCII-Hex Object Format (-a Option)

The ASCII-Hex object format supports 16-bit-bit addresses. The format consists of a byte stream with bytes separated by spaces. [Figure 11-6](#) illustrates the ASCII-Hex format.

Figure 11-6. ASCII-Hex Object Format



The file begins with an ASCII STX character (ctrl-B, 02h) and ends with an ASCII ETX character (ctrl-C, 03h). Address records are indicated with \$AXXXXXXXX, in which XXXXXXXX is a 8-digit (16-bit) hexadecimal address. The address records are present only in the following situations:

- When discontinuities occur
- When the byte stream does not begin at address 0

You can avoid all discontinuities and any address records by using the -image and -zero options. This creates output that is simply a list of byte values.

11.10.2 Intel MCS-86 Object Format (-i Option)

The Intel object format supports 16-bit addresses and 32-bit extended addresses. Intel format consists of a 9-character (4-field) prefix (which defines the start of record, byte count, load address, and record type), the data, and a 2-character checksum suffix.

The 9-character prefix represents three record types:

| Record Type | Description |
|-------------|--------------------------------|
| 00 | Data record |
| 01 | End-of-file record |
| 04 | Extended linear address record |

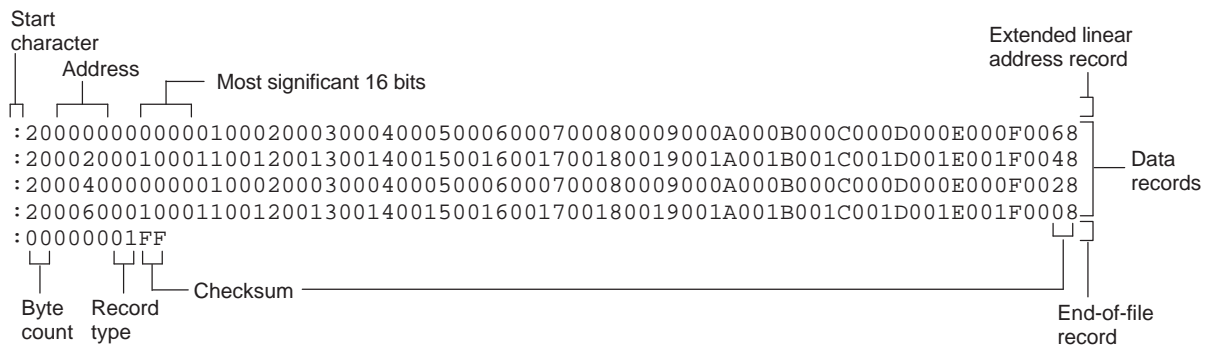
Record type 00, the data record, begins with a colon (:) and is followed by the byte count, the address of the first data byte, the record type (00), and the checksum. The address is the least significant 16 bits of a 32-bit address; this value is concatenated with the value from the most recent 04 (extended linear address) record to create a full 32-bit address. The checksum is the 2s complement (in binary form) of the preceding bytes in the record, including byte count, address, and data bytes.

Record type 01, the end-of-file record, also begins with a colon (:), followed by the byte count, the address, the record type (01), and the checksum.

Record type 04, the extended linear address record, specifies the upper 16 address bits. It begins with a colon (:), followed by the byte count, a dummy address of 0h, the record type (04), the most significant 16 bits of the address, and the checksum. The subsequent address fields in the data records contain the least significant bytes of the address.

Figure 11-7 illustrates the Intel hexadecimal object format.

Figure 11-7. Intel Hexadecimal Object Format



11.10.3 Motorola Exorciser Object Format (-m Option)

The Motorola S1, S2, and S3 formats support 16-bit, 24-bit, and 32-bit addresses, respectively. The formats consist of a start-of-file (header) record, data records, and an end-of-file (termination) record. Each record consists of five fields: record type, byte count, address, data, and checksum. The three record types are:

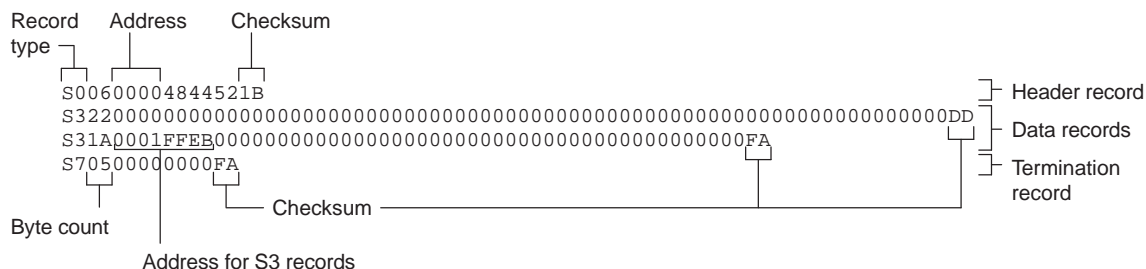
| Record Type | Description |
|-------------|---|
| S0 | Header record |
| S1 | Code/data record for 16-bit addresses (S1 format) |
| S2 | Code/data record for 24-bit addresses (S2 format) |
| S3 | Code/data record for 32-bit addresses (S3 format) |
| S7 | Termination record for 32-bit addresses (S3 format) |
| S8 | Termination record for 24-bit addresses (S2 format) |
| S9 | Termination record for 16-bit addresses (S1 format) |

The byte count is the character pair count in the record, excluding the type and byte count itself.

The checksum is the least significant byte of the 1s complement of the sum of the values represented by the pairs of characters making up the byte count, address, and the code/data fields.

Figure 11-8 illustrates the Motorola-S object format.

Figure 11-8. Motorola-S Format



11.10.4 Texas Instruments SDSMAC (TI-Tagged) Object Format (-t Option)

The Texas Instruments SDSMAC (TI-Tagged) object format supports 16-bit addresses, including start-of-file record, data records, and end-of-file record. Each data records consists of a series of small fields and is signified by a tag character:

| Tag Character | Description |
|---------------|---|
| K | Followed by the program identifier |
| 7 | Followed by a checksum |
| 8 | Followed by a dummy checksum (ignored) |
| 9 | Followed by a 16-bit load address |
| B | Followed by a data word (four characters) |
| F | Identifies the end of a data record |
| * | Followed by a data byte (two characters) |

Figure 11-9 illustrates the tag characters and fields in TI-Tagged object format.

Figure 11-9. TI-Tagged Object Format



If any data fields appear before the first address, the first field is assigned address 0000h. Address fields may be expressed but not required for any data byte. The checksum field, preceded by the tag character 7, is the 2s complement of the sum of the 8-bit ASCII values of characters, beginning with the first tag character and ending with the checksum tag character (7 or 8). The end-of-file record is a colon (:).

11.10.5 TI-TXT Hex Format (--ti_txt Option)

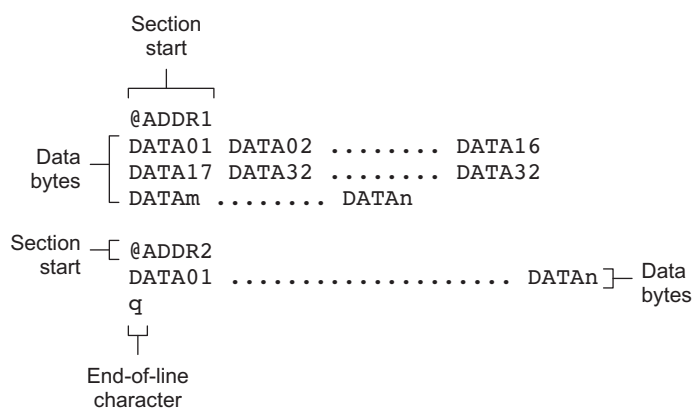
The TI-TXT hex format supports 16-bit hexadecimal data. It consists of section start addresses, data byte, and an end-of-file character. These restrictions apply:

- The number of sections is unlimited.
- Each hexadecimal start address must be even.
- Each line must have 16 data bytes, except the last line of a section.
- Data bytes are separated by a single space.
- The end-of-file termination tag q is mandatory.

The data record contains the following information:

| Item | Description |
|-------------------|--|
| @ADDR | Hexadecimal start address of a section |
| DATA _n | Hexadecimal data byte |
| q | End-of-file termination character |

Figure 11-10. TI-TXT Object Format



Example 11-3. TI-TXT Object Format

```
@F000
31 40 00 03 B2 40 80 5A 20 01 D2 D3 22 00 D2 E3
21 00 3F 40 E8 FD 1F 83 FE 23 F9 3F
@FFFE
00 F0
Q
```

11.10.6 Extended Tektronix Object Format (-x Option)

The Tektronix object format supports 32-bit addresses and has two types of records:

- Data records** contains the header field, the load address, and the object code.
- Termination records** signifies the end of a module.

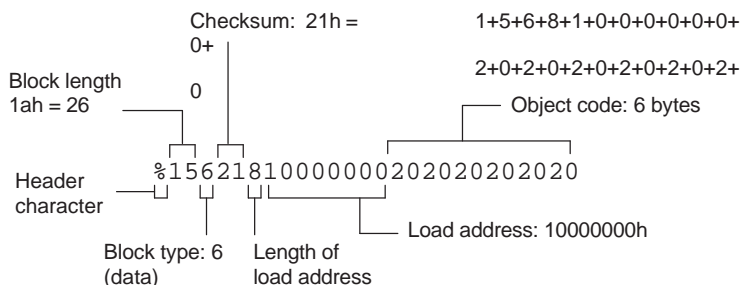
The header field in the data record contains the following information:

| Item | Number of ASCII Characters | Description |
|--------------|----------------------------|--|
| % | 1 | Data type is Tektronix format |
| Block length | 2 | Number of characters in the record, minus the % |
| Block type | 1 | 6 = data record 8 = termination record |
| Checksum | 2 | A 2-digit hex sum modulo 256 of all values in the record except the % and the checksum itself. |

The load address in the data record specifies where the object code will be located. The first digit specifies the address length; this is always 8. The remaining characters of the data record contain the object code, two characters per byte.

Figure 11-11 illustrates the Tektronix object format.

Figure 11-11. Extended Tektronix Object Format



Sharing C/C++ Header Files With Assembly Source

You can use the `.cdecls` assembler directive to share C headers containing declarations and prototypes between C and assembly code. Any legal C/C++ can be used in a `.cdecls` block and the C/C++ declarations will cause suitable assembly to be generated automatically, allowing you to reference the C/C++ constructs in assembly code.

| Topic | Page |
|---|------|
| 12.1 Overview of the <code>.cdecls</code> Directive | 256 |
| 12.2 Notes on C/C++ Conversions | 256 |
| 12.3 Notes on C++ Specific Conversions | 260 |
| 12.4 New Assembler Support..... | 261 |

12.1 Overview of the `.cdecls` Directive

The `.cdecls` directive allows programmers in mixed assembly and C/C++ environments to share C headers containing declarations and prototypes between the C and assembly code. Any legal C/C++ can be used in a `.cdecls` block and the C/C++ declarations will cause suitable assembly to be generated automatically. This allows the programmer to reference the C/C++ constructs in assembly code — calling functions, allocating space, and accessing structure members — using the equivalent assembly mechanisms. While function and variable definitions are ignored, most common C/C++ elements are converted to assembly: enumerations, (non function-like) macros, function and variable prototypes, structures, and unions.

See [the `.cdecls` topic](#) for details on the syntax of the `.cdecls` assembler directive.

The `.cdecls` directive can appear anywhere in an assembly source file, and can occur multiple times within a file. However, the C/C++ environment created by one `.cdecls` is **not** inherited by a later `.cdecls`; the C/C++ environment starts over for each `.cdecls` instance.

For example, the following code causes the warning to be issued:

```
.cdecls C,NOLIST
%{
    #define ASMTEST 1
%}

.cdecls C,NOLIST
%{
    #ifndef ASMTEST
        #warn "ASMTEST not defined!" /* will be issued */
    #endif
%}
```

Therefore, a typical use of the `.cdecls` block is expected to be a single usage near the beginning of the assembly source file, in which all necessary C/C++ header files are included.

Use the compiler `--include_path=path` options to specify additional include file paths needed for the header files used in assembly, as you would when compiling C files.

Any C/C++ errors or warnings generated by the code of the `.cdecls` are emitted as they normally would for the C/C++ source code. C/C++ errors cause the directive to fail, and any resulting converted assembly is not included.

C/C++ constructs that cannot be converted, such as function-like macros or variable definitions, cause a comment to be output to the converted assembly file. For example:

```
; ASM HEADER WARNING - variable definition 'ABCD' ignored
```

The prefix `ASM HEADER WARNING` appears at the beginning of each message. To see the warnings, either the `WARN` parameter needs to be specified so the messages are displayed on `STDERR`, or else the `LIST` parameter needs to be specified so the warnings appear in the listing file, if any.

Finally, note that the converted assembly code does not appear in the same order as the original C/C++ source code and C/C++ constructs may be simplified to a normalized form during the conversion process, but this should not affect their final usage.

12.2 Notes on C/C++ Conversions

The following sections describe C and C++ conversion elements that you need to be aware of when sharing header files with assembly source.

12.2.1 Comments

Comments are consumed entirely at the C level, and do not appear in the resulting converted assembly file.

12.2.2 Conditional Compilation (#if/#else/#ifdef/etc.)

Conditional compilation is handled entirely at the C level during the conversion step. Define any necessary macros either on the command line (using the compiler `--define=name=value` option) or within a `.cdecls` block using `#define`. The `#if`, `#ifdef`, etc. C/C++ directives are **not** converted to assembly `.if`, `.else`, `.elseif`, and `.endif` directives.

12.2.3 Pragmas

Pragmas found in the C/C++ source code cause a warning to be generated as they are not converted. They have no other effect on the resulting assembly file. See [the .cdecls topic](#) for the `WARN` and `NOWARN` parameter discussion for where these warnings are created.

12.2.4 The #error and #warning Directives

These preprocessor directives are handled completely by the compiler during the parsing step of conversion. If one of these directives is encountered, the appropriate error or warning message is emitted. These directives are not converted to `.msg` or `.wmsg` in the assembly output.

12.2.5 Predefined symbol `__ASM_HEADER__`

The C/C++ macro `__ASM_HEADER__` is defined in the compiler while processing code within `.cdecls`. This allows you to make changes in your code, such as not compiling definitions, during the `.cdecls` processing.

Be Careful With the `__ASM_HEADER__` Macro

Note: You must be very careful not to use this macro to introduce any changes in the code that could result in inconsistencies between the code processed while compiling the C/C++ source and while converting to assembly.

12.2.6 Usage Within C/C++ `asm()` Statements

The `.cdecls` directive is not allowed within C/C++ `asm()` statements and will cause an error to be generated.

12.2.7 The #include Directive

The C/C++ `#include` preprocessor directive is handled transparently by the compiler during the conversion step. Such `#includes` can be nested as deeply as desired as in C/C++ source. The assembly directives `.include` and `.copy` are not used or needed within a `.cdecls`. Use the command line `--include_path` option to specify additional paths to be searched for included files, as you would for C compilation.

12.2.8 Conversion of #define Macros

Only object-like macros are converted to assembly. Function-like macros have no assembly representation and so cannot be converted. Pre-defined and built-in C/C++ macros are not converted to assembly (i.e., `__FILE__`, `__TIME__`, `__TI_COMPILER_VERSION__`, etc.). For example, this code is converted to assembly because it is an object-like macro:

```
#define NAME Charley
```

This code is not converted to assembly because it is a function-like macro:

```
#define MAX(x,y) (x>y ? x : y)
```

Some macros, while they are converted, have no functional use in the containing assembly file. For example, the following results in the assembly substitution symbol `FOREVER` being set to the value `while(1)`, although this has no useful use in assembly because `while(1)` is not legal assembly code.

```
#define FOREVER while(1)
```

Macro values are **not** interpreted as they are converted. For example, the following results in the assembler substitution symbol `OFFSET` being set to the literal string value `5+12` and **not** the value 17. This happens because the semantics of the C/C++ language require that macros are evaluated in context and not when they are parsed.

```
#define OFFSET 5+12
```

Because macros in C/C++ are evaluated in their usage context, C/C++ printf escape sequences such as `\n` are not converted to a single character in the converted assembly macro. See [Section 12.2.11](#) for suggestions on how to use C/C++ macro strings.

Macros are converted using the new `.define` directive (see [Section 12.4.2](#)), which functions similarly to the `.asg` assembler directive. The exception is that `.define` disallows redefinitions of register symbols and mnemonics to prevent the conversion from corrupting the basic assembly environment. To remove a macro from the assembly scope, `.undef` can be used following the `.cdecls` that defines it (see [Section 12.4.3](#)).

The macro functionality of `#` (stringize operator) is only useful within functional macros. Since functional macros are not supported by this process, `#` is not supported either. The concatenation operator `##` is only useful in a functional context, but can be used degenerately to concatenate two strings and so it is supported in that context.

12.2.9 The #undef Directive

Symbols undefined using the `#undef` directive before the end of the `.cdecls` are not converted to assembly.

12.2.10 Enumerations

Enumeration members are converted to `.enum` elements in assembly. For example:

```
enum state { ACTIVE=0x10, SLEEPING=0x01, INTERRUPT=0x100, POWEROFF, LAST};
```

is converted to the following assembly code:

```
state      .enum
ACTIVE    .emember 16
SLEEPING  .emember 1
INTERRUPT .emember 256
POWEROFF  .emember 257
LAST      .emember 258
          .endenum
```

The members are used via the pseudo-scoping created by the `.enum` directive.

The usage is similar to that for accessing structure members, `enum_name.member`.

This pseudo-scoping is used to prevent enumeration member names from corrupting other symbols within the assembly environment.

12.2.11 C Strings

Because C string escapes such as `\n` and `\t` are not converted to hex characters `0x0A` and `0x09` until their use in a string constant in a C/C++ program, C macros whose values are strings cannot be represented as expected in assembly substitution symbols. For example:

```
#define MSG "\tHI\n"
```

becomes, in assembly:

```
.define "" "\tHI\n",MSG ; 6 quoted characters! not 5!
```

When used in a C string context, you expect this statement to be converted to 5 characters (tab, H, I, newline, NULL), but the `.string` assembler directive does not know how to perform the C escape conversions.

You can use the `.cstring` directive to cause the escape sequences and NULL termination to be properly handled as they would in C/C++. Using the above symbol `MSG` with a `.cstring` directive results in 5 characters of memory being allocated, the same characters as would result if used in a C/C++ string context. (See [Section 12.4.7](#) for the `.cstring` directive syntax.)

12.2.12 C/C++ Built-In Functions

The C/C++ built-in functions, such as `sizeof()`, are not translated to their assembly counterparts, if any, if they are used in macros. Also, their C expression values are not inserted into the resulting assembly macro because macros are evaluated in context and there is no active context when converting the macros to assembly.

Suitable functions such as `$_sizeof()` are available in assembly expressions. However, as the basic types such as `int/char/float` have no type representation in assembly, there is no way to ask for `$_sizeof(int)`, for example, in assembly.

12.2.13 Structures and Unions

C/C++ structures and unions are converted to assembly `.struct` and `.union` elements. Padding and ending alignments are added as necessary to make the resulting assembly structure have the same size and member offsets as the C/C++ source. The primary purpose is to allow access to members of C/C++ structures, as well as to facilitate debugging of the assembly code. For nested structures, the assembly `.tag` feature is used to refer to other structures/unions.

The alignment is also passed from the C/C++ source so that the assembly symbol is marked with the same alignment as the C/C++ symbol. (See [Section 12.2.3](#) for information about pragmas, which may attempt to modify structures.) Because the alignment of structures is stored in the assembly symbol, built-in assembly functions like `$_sizeof()` and `$_alignof()` can be used on the resulting structure name symbol.

When using unnamed structures (or unions) in typedefs, such as:

```
typedef struct { int a_member; } mystrname;
```

This is really a shorthand way of writing:

```
struct temporary_name { int a_member; };
typedef temporary_name mystrname;
```

The conversion processes the above statements in the same manner: generating a temporary name for the structure and then using `.define` to output a typedef from the temporary name to the user name. You should use your *mystrname* in assembly the same as you would in C/C++, but do not be confused by the assembly structure definition in the list, which contains the temporary name. You can avoid the temporary name by specifying a name for the structure, as in:

```
typedef struct a_st_name { ... } mystrname;
```

If a shorthand method is used in C to declare a variable with a particular structure, for example:

```
extern struct a_name { int a_member; } a_variable;
```

Then after the structure is converted to assembly, a `.tag` directive is generated to declare the structure of the external variable, such as:

```
_a_variable .tag a_st_name
```

This allows you to refer to `_a_variable.a_member` in your assembly code.

12.2.14 Function/Variable Prototypes

Non-static function and variable prototypes (not definitions) will result in a `.global` directive being generated for each symbol found.

See [Section 12.3.1](#) for C++ name mangling issues.

Function and variable definitions will result in a warning message being generated (see the `WARN/NOWARN` parameter discussion for where these warnings are created) for each, and they will not be represented in the converted assembly.

The assembly symbol representing the variable declarations will not contain type information about those symbols. Only a `.global` will be issued for them. Therefore, it is your responsibility to ensure the symbol is used appropriately.

See [Section 12.2.13](#) for information on variables names which are of a structure/union type.

12.2.15 C Constant Suffixes

The C constant suffixes u, l, and f are passed to the assembly unchanged. The assembler will ignore these suffixes if used in assembly expressions.

12.2.16 Basic C/C++ Types

Only complex types (structures and unions) in the C/C++ source code are converted to assembly. Basic types such as int, char, or float are not converted or represented in assembly beyond any existing .int, .char, .float, etc. directives that previously existed in assembly.

Typedefs of basic types are therefore also not represented in the converted assembly.

12.3 Notes on C++ Specific Conversions

The following sections describe C++ specific conversion elements that you need to be aware of when sharing header files with assembly source.

12.3.1 Name Mangling

Symbol names may be mangled in C++ source files. When mangling occurs, the converted assembly will use the mangled names to avoid symbol name clashes. You can use the demangler (dem430) to demangle names and identify the correct symbols to use in assembly.

To defeat name mangling in C++ for symbols where polymorphism (calling a function of the same name with different kinds of arguments) is not required, use the following syntax:

```
extern "C" void somefunc(int arg);
```

The above format is the short method for declaring a single function. To use this method for multiple functions, you can also use the following syntax:

```
extern "C"
{
    void somefunc(int arg);
    int  anotherfunc(int arg);
    ...
}
```

12.3.2 Derived Classes

Derived classes are only partially supported when converting to assembly because of issues related to C++ scoping which does not exist in assembly. The greatest difference is that base class members do not automatically become full (top-level) members of the derived class. For example:

```
-----
class base
{
    public:
        int b1;
};

class derived : public base
{
    public:
        int d1;
}
-----
```

In C++ code, the class derived would contain both integers b1 and d1. In the converted assembly structure "derived", the members of the base class must be accessed using the name of the base class, such as derived.__b_base.b1 rather than the expected derived.b1.

A non-virtual, non-empty base class will have __b_ prepended to its name within the derived class to signify it is a base class name. That is why the example above is derived.__b_base.b1 and not simply derived.base.b1.

12.3.3 Templates

No support exists for templates.

12.3.4 Virtual Functions

No support exists for virtual functions, as they have no assembly representation.

12.4 New Assembler Support

12.4.1 Enumerations (*.enum/.emember/.endenum*)

New directives have been created to support a pseudo-scoping for enumerations.

The format of these new directives is:

```

ENUM_NAME      .enum
MEMBER1        .emember [value]
MEMBER2        .emember [value]
...
                .endenum
  
```

The **.enum** directive begins the enumeration definition and **.endenum** terminates it.

The enumeration name (*ENUM_NAME*) cannot be used to allocate space; its size is reported as zero.

The format to use the value of a member is *ENUM_NAME.MEMBER*, similar to a structure member usage.

The **.emember** directive optionally accepts the value to set the member to, just as in C/C++. If not specified, the member takes a value one more than the previous member. As in C/C++, member names cannot be duplicated, although values can be. Unless specified with **.emember**, the first enumeration member will be given the value 0 (zero), as in C/C++.

The **.endenum** directive cannot be used with a label, as structure **.endstruct** directives can, because the **.endenum** directive has no value like the **.endstruct** does (containing the size of the structure).

Conditional compilation directives (*.if/.else/.elseif/.endif*) are the only other non-enumeration code allowed within the *.enum/.endenum* sequence.

12.4.2 The *.define* Directive

The new **.define** directive functions in the same manner as the existing **.asg** directive, except that **.define** disallows creation of a substitution symbol that has the same name as a register symbol or mnemonic. It does not create a new symbol name space in the assembler, rather it uses the existing substitution symbol name space. The syntax for the directive is:

```
.define substitution string, substitution symbol name
```

The **.define** directive is used to prevent corruption of the assembly environment when converting C/C++ headers.

12.4.3 The *.undef/.unasg* Directives

The **.undef** directive is used to remove the definition of a substitution symbol created using **.define** or **.asg**. This directive will remove the named symbol from the substitution symbol table from the point of the **.undef** to the end of the assembly file. The syntax for these directives is:

```
.undefine substitution symbol name
```

.unasg *substitution symbol name*

This can be used to remove from the assembly environment any C/C++ macros that may cause a problem.

Also see [Section 12.4.2](#), which covers the `.define` directive.

12.4.4 The `$defined()` Directive

The `$defined` directive returns true/1 or false/0 depending on whether the name exists in the current substitution symbol table or the standard symbol table. In essence `$defined` returns TRUE if the assembler has any user symbol in scope by that name. This differs from `$isdefed` in that `$isdefed` only tests for NON-substitution symbols. The syntax is:

\$defined(*substitution symbol name***)**

A statement such as `.if $defined(macroname)` is then similar to the C code `#ifdef macroname`.

See [Section 12.4.2](#) and [Section 12.4.3](#) for the use of `.define` and `.undef` in assembly.

12.4.5 The `$sizeof` Built-In Function

The new assembly built-in function `$sizeof()` can be used to query the size of a structure in assembly. It is an alias for the already existing `$structsz()`. The syntax is:

\$sizeof(*structure name***)**

The `$sizeof` function can then be used similarly to the C built-in function `sizeof()`.

The assembler's `$sizeof()` built-in function cannot be used to ask for the size of basic C/C++ types, such as `$sizeof(int)`, because those basic type names are not represented in assembly. Only complex types are converted from C/C++ to assembly.

Also see [Section 12.2.12](#), which notes that this conversion does not happen automatically if the C/C++ `sizeof()` built-in function is used within a macro.

12.4.6 Structure/Union Alignment & `$alignof()`

The assembly `.struct` and `.union` directives now take an optional second argument which can be used to specify a minimum alignment to be applied to the symbol name. This is used by the conversion process to pass the specific alignment from C/C++ to assembly.

The assembly built-in function `$alignof()` can be used to report the alignment of these structures. This can be used even on assembly structures, and the function will return the minimum alignment calculated by the assembler.

12.4.7 The `.cstring` Directive

You can use the new `.cstring` directive to cause the escape sequences and NULL termination to be properly handled as they would in C/C++.

```
.cstring "String with C escapes.\nWill be NULL terminated.\012"
```

See [Section 12.2.11](#) for more information on the new `.cstring` directive.

Symbolic Debugging Directives

The assembler supports several directives that the MSP430 C/C++ compiler uses for symbolic debugging. These directives differ for the two debugging formats, DWARF and COFF.

These directives are not meant for use by assembly-language programmers. They require arguments that can be difficult to calculate manually, and their usage must conform to a predetermined agreement between the compiler, the assembler, and the debugger. This appendix documents these directives for informational purposes only.

| Topic | Page |
|---------------------------------|------|
| A.1 DWARF Debugging Format..... | 264 |
| A.2 COFF Debugging Format..... | 264 |
| A.3 Debug Directive Syntax..... | 265 |

A.1 DWARF Debugging Format

A subset of the DWARF symbolic debugging directives are always listed in the assembly language file that the compiler creates for program analysis purposes. To list the complete set used for full symbolic debug, invoke the compiler with the `--symdebug:dwarf` option, as shown below:

```
cl430 --symdebug:dwarf --keep_asm input_file
```

The `--keep_asm` option instructs the compiler to retain the generated assembly file.

To disable the generation of all symbolic debug directives, invoke the compiler with the `-symdebug:none` option:

```
cl430 --symdebug:none --keep_asm input_file
```

The DWARF debugging format consists of the following directives:

- The **.dwtag** and **.dwendtag** directives define a Debug Information Entry (DIE) in the `.debug_info` section.
- The **.dwattr** directive adds an attribute to an existing DIE.
- The **.dwpsn** directive identifies the source position of a C/C++ statement.
- The **.dwcie** and **.dwentry** directives define a Common Information Entry (CIE) in the `.debug_frame` section.
- The **.dwfde** and **.dwentry** directives define a Frame Description Entry (FDE) in the `.debug_frame` section.
- The **.dwcfi** directive defines a call frame instruction for a CIE or FDE.

A.2 COFF Debugging Format

COFF symbolic debug is now obsolete. These directives are supported for backwards-compatibility only. The decision to switch to DWARF as the symbolic debug format was made to overcome many limitations of COFF symbolic debug, including the absence of C++ support.

The COFF debugging format consists of the following directives:

- The **.sym** directive defines a global variable, a local variable, or a function. Several parameters allow you to associate various debugging information with the variable or function.
- The **.stag**, **.etag**, and **.utag** directives define structures, enumerations, and unions, respectively. The **.member** directive specifies a member of a structure, enumeration, or union. The **.eos** directive ends a structure, enumeration, or union definition.
- The **.func** and **.endfunc** directives specify the beginning and ending lines of a C/C++ function.
- The **.block** and **.endblock** directives specify the bounds of C/C++ blocks.
- The **.file** directive defines a symbol in the symbol table that identifies the current source filename.
- The **.line** directive identifies the line number of a C/C++ source statement.

A.3 Debug Directive Syntax

Table A-1 is an alphabetical listing of the symbolic debugging directives. For information on the C/C++ compiler, refer to the *MSP430 Optimizing C/C++ Compiler User's Guide*

Table A-1. Symbolic Debugging Directives

| Label | Directive | Arguments |
|-----------|------------------|---|
| | .block | [beginning line number] |
| | .dwattr | DIE label,DIE attribute name(DIE attribute value)[,DIE attribute name(attribute value) [, ...] |
| | .dwcfi | call frame instruction opcode[,operand[,operand]] |
| CIE label | .dwcie | version , return address register |
| | .dwentry | |
| | .dwendtag | |
| | .dwfde | CIE label |
| | .dwpsn | " filename " , line number , column number |
| DIE label | .dwtag | DIE tag name,DIE attribute name(DIE attribute value)[,DIE attribute name(attribute value) [, ...] |
| | .endblock | [ending line number] |
| | .endfunc | [ending line number[,register mask[, frame size]]] |
| | .eos | |
| | .etag | name[, size] |
| | .file | " filename " |
| | .func | [beginning line number] |
| | .line | line number[, address] |
| | .member | name, value[, type, storage class, size, tag, dims] |
| | .stag | name[, size] |
| | .sym | name, value[, type, storage class, size, tag, dims] |
| | .utag | name[, size] |

XML Link Information File Description

The MSP430 linker supports the generation of an XML link information file via the `--xml_link_info file` option. This option causes the linker to generate a well-formed XML file containing detailed information about the result of a link. The information included in this file includes all of the information that is currently produced in a linker-generated map file.

As the linker evolves, the XML link information file may be extended to include additional information that could be useful for static analysis of linker results.

This appendix enumerates all of the elements that are generated by the linker into the XML link information file.

| Topic | Page |
|---|------------|
| B.1 XML Information File Element Types | 268 |
| B.2 Document Elements..... | 268 |

B.1 XML Information File Element Types

These element types will be generated by the linker:

- **Container elements** represent an object that contains other elements that describe the object. Container elements have an id attribute that makes them accessible from other elements.
- **String elements** contain a string representation of their value.
- **Constant elements** contain a 32-bit unsigned long representation of their value (with a 0x prefix).
- **Reference elements** are empty elements that contain an idref attribute that specifies a link to another container element.

In [Section B.2](#), the element type is specified for each element in parentheses following the element description. For instance, the <link_time> element lists the time of the link execution (string).

B.2 Document Elements

The root element, or the document element, is **<link_info>**. All other elements contained in the XML link information file are children of the <link_info> element. The following sections describe the elements that an XML information file can contain.

B.2.1 Header Elements

The first elements in the XML link information file provide general information about the linker and the link session:

- The **<banner>** element lists the name of the executable and the version information (string).
- The **<copyright>** element lists the TI copyright information (string).
- The **<link_time>** is a timestamp representation of the link time (unsigned 32-bit int).
- The **<output_file>** element lists the name of the linked output file generated (string).
- The **<entry_point>** element specifies the program entry point, as determined by the linker (container) with two entries:
 - The **<name>** is the entry point symbol name, if any (string).
 - The **<address>** is the entry point address (constant).

Example B-1. Header Element for the hi.out Output File

```
<banner>TMS320Cxx Linker          Version x.xx (Jan 6 2008)</banner>
<copyright>Copyright (c) 1996-2008 Texas Instruments Incorporated</copyright>
<link_time>0x43dfd8a4</link_time>
<output_file>hi.out</output_file>
<entry_point>
  <name>_c_int00</name>
  <address>0xaf80</address>
</entry_point>
```

B.2.2 Input File List

The next section of the XML link information file is the input file list, which is delimited with a `<input_file_list>` container element. The `<input_file_list>` can contain any number of `<input_file>` elements.

Each `<input_file>` instance specifies the input file involved in the link. Each `<input_file>` has an `id` attribute that can be referenced by other elements, such as an `<object_component>`. An `<input_file>` is a container element enclosing the following elements:

- The `<path>` element names a directory path, if applicable (string).
- The `<kind>` element specifies a file type, either archive or object (string).
- The `<file>` element specifies an archive name or filename (string).
- The `<name>` element specifies an object file name, or archive member name (string).

Example B-2. Input File List for the `hi.out` Output File

```

<input_file_list>
  <input_file id="f1-1">
    <kind>object</kind>
    <file>hi.obj</file>
    <name>hi.obj</name>
  </input_file>
  <input_file id="f1-2">
    <path>/tools/lib/</path>
    <kind>archive</kind>
    <file>rtsxxx.lib</file>
    <name>boot.obj</name>
  </input_file>
  <input_file id="f1-3">
    <path>/tools/lib/</path>
    <kind>archive</kind>
    <file>rtsxxx.lib</file>
    <name>exit.obj</name>
  </input_file>
  <input_file id="f1-4">
    <path>/tools/lib/</path>
    <kind>archive</kind>
    <file>rtsxxx.lib</file>
    <name>printf.obj</name>
  </input_file>
  ...
</input_file_list>

```

B.2.3 Object Component List

The next section of the XML link information file contains a specification of all of the object components that are involved in the link. An example of an object component is an input section. In general, an object component is the smallest piece of object that can be manipulated by the linker.

The **<object_component_list>** is a container element enclosing any number of **<object_component>** elements.

Each **<object_component>** specifies a single object component. Each **<object_component>** has an **id** attribute so that it can be referenced directly from other elements, such as a **<logical_group>**. An **<object_component>** is a container element enclosing the following elements:

- The **<name>** element names the object component (string).
- The **<load_address>** element specifies the load-time address of the object component (constant).
- The **<run_address>** element specifies the run-time address of the object component (constant).
- The **<size>** element specifies the size of the object component (constant).
- The **<input_file_ref>** element specifies the source file where the object component originated (reference).

Example B-3. Object Component List for the fl-4 Input File

```

<object_component id="oc-20">
  <name>.text</name>
  <load_address>0xac00</load_address>
  <run_address>0xac00</run_address>
  <size>0xc0</size>
  <input_file_ref idref="fl-4"/>
</object_component>
<object_component id="oc-21">
  <name>.data</name>
  <load_address>0x80000000</load_address>
  <run_address>0x80000000</run_address>
  <size>0x0</size>
  <input_file_ref idref="fl-4"/>
</object_component>
<object_component id="oc-22">
  <name>.bss</name>
  <load_address>0x80000000</load_address>
  <run_address>0x80000000</run_address>
  <size>0x0</size>
  <input_file_ref idref="fl-4"/>
</object_component>

```

B.2.4 Logical Group List

The **<logical_group_list>** section of the XML link information file is similar to the output section listing in a linker-generated map file. However, the XML link information file contains a specification of GROUP and UNION output sections, which are not represented in a map file. There are three kinds of list items that can occur in a **<logical_group_list>**:

- The **<logical_group>** is the specification of a section or GROUP that contains a list of object components or logical group members. Each **<logical_group>** element is given an id so that it may be referenced from other elements. Each **<logical_group>** is a container element enclosing the following elements:
 - The **<name>** element names the logical group (string).
 - The **<load_address>** element specifies the load-time address of the logical group (constant).
 - The **<run_address>** element specifies the run-time address of the logical group (constant).
 - The **<size>** element specifies the size of the logical group (constant).
 - The **<contents>** element lists elements contained in this logical group (container). These elements refer to each of the member objects contained in this logical group:
 - The **<object_component_ref>** is an object component that is contained in this logical group (reference).
 - The **<logical_group_ref>** is a logical group that is contained in this logical group (reference).
- The **<overlay>** is a special kind of logical group that represents a UNION, or a set of objects that share the same memory space (container). Each **<overlay>** element is given an id so that it may be referenced from other elements (like from an **<allocated_space>** element in the placement map). Each **<overlay>** contains the following elements:
 - The **<name>** element names the overlay (string).
 - The **<run_address>** element specifies the run-time address of overlay (constant).
 - The **<size>** element specifies the size of logical group (constant).
 - The **<contents>** container element lists elements contained in this overlay. These elements refer to each of the member objects contained in this logical group:
 - The **<object_component_ref>** is an object component that is contained in this logical group (reference).
 - The **<logical_group_ref>** is a logical group that is contained in this logical group (reference).
- The **<split_section>** is another special kind of logical group that represents a collection of logical groups that is split among multiple memory areas. Each **<split_section>** element is given an id so that it may be referenced from other elements. The id consists of the following elements.
 - The **<name>** element names the split section (string).
 - The **<contents>** container element lists elements contained in this split section. The **<logical_group_ref>** elements refer to each of the member objects contained in this split section, and each element referenced is a logical group that is contained in this split section (reference).

Example B-4. Logical Group List for the fl-4 Input File

```

<logical_group_list>
  ...
  <logical_group id="lg-7">
    <name>.text</name>
    <load_address>0x20</load_address>
    <run_address>0x20</run_address>
    <size>0xb240</size>
    <contents>
      <object_component_ref idref="oc-34"/>
      <object_component_ref idref="oc-108"/>
      <object_component_ref idref="oc-e2"/>
      ...
    </contents>
  </logical_group>
  ...
  <overlay id="lg-b">
    <name>UNION_1</name>
    <run_address>0xb600</run_address>
    <size>0xc0</size>
    <contents>
      <object_component_ref idref="oc-45"/>
      <logical_group_ref idref="lg-8"/>
    </contents>
  </overlay>
  ...
  <split_section id="lg-12">
    <name>.task_scn</name>
    <size>0x120</size>
    <contents>
      <logical_group_ref idref="lg-10"/>
      <logical_group_ref idref="lg-11"/>
    </contents>
    ...
  </split_section>
</logical_group_list>

```


B.2.5 Placement Map

The **<placement_map>** element describes the memory placement details of all named memory areas in the application, including unused spaces between logical groups that have been placed in a particular memory area.

The **<memory_area>** is a description of the placement details within a named memory area (container). The description consists of these items:

- The **<name>** names the memory area (string).
- The **<page_id>** gives the id of the memory page in which this memory area is defined (constant).
- The **<origin>** specifies the beginning address of the memory area (constant).
- The **<length>** specifies the length of the memory area (constant).
- The **<used_space>** specifies the amount of allocated space in this area (constant).
- The **<unused_space>** specifies the amount of available space in this area (constant).
- The **<attributes>** lists the RWXI attributes that are associated with this area, if any (string).
- The **<fill_value>** specifies the fill value that is to be placed in unused space, if the fill directive is specified with the memory area (constant).
- The **<usage_details>** lists details of each allocated or available fragment in this memory area. If the fragment is allocated to a logical group, then a **<logical_group_ref>** element is provided to facilitate access to the details of that logical group. All fragment specifications include **<start_address>** and **<size>** elements.
 - The **<allocated_space>** element provides details of an allocated fragment within this memory area (container):
 - The **<start_address>** specifies the address of the fragment (constant).
 - The **<size>** specifies the size of the fragment (constant).
 - The **<logical_group_ref>** provides a reference to the logical group that is allocated to this fragment (reference).
 - The **<available_space>** element provides details of an available fragment within this memory area (container):
 - The **<start_address>** specifies the address of the fragment (constant).
 - The **<size>** specifies the size of the fragment (constant).

Example B-5. Placement Map for the fl-4 Input File

```

<placement_map>
  <memory_area>
    <name>PMEM</name>
    <page_id>0x0</page_id>
    <origin>0x20</origin>
    <length>0x100000</length>
    <used_space>0xb240</used_space>
    <unused_space>0xf4dc0</unused_space>
    <attributes>RWXI</attributes>
    <usage_details>
      <allocated_space>
        <start_address>0x20</start_address>
        <size>0xb240</size>
        <logical_group_ref idref="lg-7"/>
      </allocated_space>
      <available_space>
        <start_address>0xb260</start_address>
        <size>0xf4dc0</size>
      </available_space>
    </usage_details>
  </memory_area>
  ...
</placement_map>

```

B.2.6 Symbol Table

The **<symbol_table>** contains a list of all of the global symbols that are included in the link. The list provides information about a symbol's name and value. In the future, the symbol_table list may provide type information, the object component in which the symbol is defined, storage class, etc.

The **<symbol>** is a container element that specifies the name and value of a symbol with these elements:

- The **<name>** element specifies the symbol name (string).
- The **<value>** element specifies the symbol value (constant).

Example B-6. Symbol Table for the fl-4 Input File

```

<symbol_table>
  <symbol>
    <name>_c_int00</name>
    <value>0xaf80</value>
  </symbol>
  <symbol>
    <name>_main</name>
    <value>0xb1e0</value>
  </symbol>
  <symbol>
    <name>_printf</name>
    <value>0xac00</value>
  </symbol>
  ...
</symbol_table>

```

Glossary

- absolute address**—An address that is permanently assigned to a MSP430 memory location.
- absolute lister**—A debugging tool that allows you to create assembler listings that contain absolute addresses.
- alignment**— A process in which the linker places an output section at an address that falls on an n -byte boundary, where n is a power of 2. You can specify alignment with the `SECTIONS` linker directive.
- allocation**— A process in which the linker calculates the final memory addresses of output sections.
- ANSI**— American National Standards Institute; an organization that establishes standards voluntarily followed by industries.
- archive library**—A collection of individual files grouped into a single file by the archiver.
- archiver**— A software program that collects several individual files into a single file called an archive library. With the archiver, you can add, delete, extract, or replace members of the archive library.
- ASCII**— American Standard Code for Information Interchange; a standard computer code for representing and exchanging alphanumeric information.
- assembler**— A software program that creates a machine-language program from a source file that contains assembly language instructions, directives, and macro definitions. The assembler substitutes absolute operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.
- assembly-time constant**— A symbol that is assigned a constant value with the `.set` directive.
- big endian**—An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *little endian*.
- binding**— A process in which you specify a distinct address for an output section or a symbol.
- BIS**— Bit instruction set.
- block**— A set of statements that are grouped together within braces and treated as an entity.
- .bss section**—One of the default object file sections. You use the assembler `.bss` directive to reserve a specified amount of space in the memory map that you can use later for storing data. The `.bss` section is uninitialized.
- byte**— Per ANSI/ISO C, the smallest addressable unit that can hold a character.
- C/C++ compiler**—A software program that translates C source statements into assembly language source statements.
- COFF**— Common object file format; a system of object files configured according to a standard developed by AT&T. These files are relocatable in memory space.
- command file**—A file that contains options, filenames, directives, or commands for the linker or hex conversion utility.

- comment**— A source statement (or portion of a source statement) that documents or improves readability of a source file. Comments are not compiled, assembled, or linked; they have no effect on the object file.
- compiler program**— A utility that lets you compile, assemble, and optionally link in one step. The compiler runs one or more source modules through the compiler (including the parser, optimizer, and code generator), the assembler, and the linker.
- conditional processing**— A method of processing one block of source code or an alternate block of source code, according to the evaluation of a specified expression.
- configured memory**— Memory that the linker has specified for allocation.
- constant**— A type whose value cannot change.
- cross-reference lister**— A utility that produces an output file that lists the symbols that were defined, what file they were defined in, what reference type they are, what line they were defined on, which lines referenced them, and their assembler and linker final values. The cross-reference lister uses linked object files as input.
- cross-reference listing**— An output file created by the assembler that lists the symbols that were defined, what line they were defined on, which lines referenced them, and their final values.
- .data section**— One of the default object file sections. The .data section is an initialized section that contains initialized data. You can use the .data directive to assemble code into the .data section.
- directives**— Special-purpose commands that control the actions and functions of a software tool (as opposed to assembly language instructions, which control the actions of a device).
- ELF**— Executable and linking format; a system of object files configured according to the System V Application Binary Interface specification.
- emulator**— A hardware development system that duplicates the MSP430 operation.
- entry point**— A point in target memory where execution starts.
- environment variable**— A system symbol that you define and assign to a string. Environmental variables are often included in Windows batch files or UNIX shell scripts such as .cshrc or .profile.
- epilog**— The portion of code in a function that restores the stack and returns.
- executable module**— A linked object file that can be executed in a target system.
- expression**— A constant, a symbol, or a series of constants and symbols separated by arithmetic operators.
- external symbol**— A symbol that is used in the current program module but defined or declared in a different program module.
- field**— For the MSP430, a software-configurable data type whose length can be programmed to be any value in the range of 1-16 bits.
- global symbol**— A symbol that is either defined in the current module and accessed in another, or accessed in the current module but defined in another.
- GROUP**— An option of the SECTIONS directive that forces specified output sections to be allocated contiguously (as a group).
- hex conversion utility**— A utility that converts object files into one of several standard ASCII hexadecimal formats, suitable for loading into an EPROM programmer.
- high-level language debugging**— The ability of a compiler to retain symbolic and high-level language information (such as type and function definitions) so that a debugging tool can use this information.

- hole**— An area between the input sections that compose an output section that contains no code.
- incremental linking**— Linking files in several passes. Incremental linking is useful for large applications, because you can partition the application, link the parts separately, and then link all of the parts together.
- initialization at load time**—An autoinitialization method used by the linker when linking C/C++ code. The linker uses this method when you invoke it with the `--ram_model` link option. This method initializes variables at load time instead of run time.
- initialized section**—A section from an object file that will be linked into an executable module.
- input section**—A section from an object file that will be linked into an executable module.
- ISO**— International Organization for Standardization; a worldwide federation of national standards bodies, which establishes international standards voluntarily followed by industries.
- label**— A symbol that begins in column 1 of an assembler source statement and corresponds to the address of that statement. A label is the only assembler statement that can begin in column 1.
- linker**— A software program that combines object files to form an object module that can be allocated into system memory and executed by the device.
- listing file**—An output file, created by the assembler, that lists source statements, their line numbers, and their effects on the section program counter (SPC).
- little endian**—An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *big endian*
- loader**— A device that places an executable module into system memory.
- macro**— A user-defined routine that can be used as an instruction.
- macro call**—The process of invoking a macro.
- macro definition**—A block of source statements that define the name and the code that make up a macro.
- macro expansion**—The process of inserting source statements into your code in place of a macro call.
- macro library**— An archive library composed of macros. Each file in the library must contain one macro; its name must be the same as the macro name it defines, and it must have an extension of `.asm`.
- map file**—An output file, created by the linker, that shows the memory configuration, section composition, section allocation, symbol definitions and the addresses at which the symbols were defined for your program.
- member**— The elements or variables of a structure, union, archive, or enumeration.
- memory map**—A map of target system memory space that is partitioned into functional blocks.
- mnemonic**— An instruction name that the assembler translates into machine code.
- model statement**— Instructions or assembler directives in a macro definition that are assembled each time a macro is invoked.
- named section**— An initialized section that is defined with a `.sect` directive.
- object file**—An assembled or linked file that contains machine-language object code.
- object library**—An archive library made up of individual object files.
- object module**—A linked, executable object file that can be downloaded and executed on a target system.

- operand**— An argument of an assembly language instruction, assembler directive, or macro directive that supplies information to the operation performed by the instruction or directive.
- optimizer**— A software tool that improves the execution speed and reduces the size of C programs.
- options**— Command-line parameters that allow you to request additional or specific functions when you invoke a software tool.
- output module**—A linked, executable object file that is downloaded and executed on a target system.
- output section**—A final, allocated section in a linked, executable module.
- partial linking**— Linking files in several passes. Incremental linking is useful for large applications because you can partition the application, link the parts separately, and then link all of the parts together.
- quiet run**— An option that suppresses the normal banner and the progress information.
- raw data**—Executable code or initialized data in an output section.
- relocation**— A process in which the linker adjusts all the references to a symbol when the symbol's address changes.
- ROM width**—The width (in bits) of each output file, or, more specifically, the width of a single data value in the hex conversion utility file. The ROM width determines how the utility partitions the data into output files. After the target words are mapped to memory words, the memory words are broken into one or more output files. The number of output files is determined by the ROM width.
- run address**—The address where a section runs.
- run-time-support library**—A library file, `rts.src`, that contains the source for the run time-support functions.
- section**— A relocatable block of code or data that ultimately will be contiguous with other sections in the memory map.
- section program counter (SPC)**— An element that keeps track of the current location within a section; each section has its own SPC.
- sign extend**—A process that fills the unused MSBs of a value with the value's sign bit.
- simulator**— A software development system that simulates MSP430 operation.
- source file**—A file that contains C/C++ code or assembly language code that is compiled or assembled to form an object file.
- static variable**—A variable whose scope is confined to a function or a program. The values of static variables are not discarded when the function or program is exited; their previous value is resumed when the function or program is reentered.
- storage class**—An entry in the symbol table that indicates how to access a symbol.
- string table**—A table that stores symbol names that are longer than eight characters (symbol names of eight characters or longer cannot be stored in the symbol table; instead they are stored in the string table). The name portion of the symbol's entry points to the location of the string in the string table.
- structure**— A collection of one or more variables grouped together under a single name.
- subsection**— A relocatable block of code or data that ultimately will occupy continuous space in the memory map. Subsections are smaller sections within larger sections. Subsections give you tighter control of the memory map.
- symbol**— A string of alphanumeric characters that represents an address or a value.
- symbolic debugging**—The ability of a software tool to retain symbolic information that can be used by a debugging tool such as a simulator or an emulator.

- tag**— An optional *type* name that can be assigned to a structure, union, or enumeration.
- target memory**— Physical memory in a system into which executable object code is loaded.
- .text section**—One of the default object file sections. The .text section is initialized and contains executable code. You can use the .text directive to assemble code into the .text section.
- unconfigured memory**— Memory that is not defined as part of the memory map and cannot be loaded with code or data.
- uninitialized section**—A object file section that reserves space in the memory map but that has no actual contents. These sections are built with the .bss and .usect directives.
- UNION**— An option of the SECTIONS directive that causes the linker to allocate the same address to multiple sections.
- union**— A variable that can hold objects of different types and sizes.
- unsigned value**—A value that is treated as a nonnegative number, regardless of its actual sign.
- variable**— A symbol representing a quantity that can assume any of a set of values.
- vener**— A sequence of instructions that serves as an alternate entry point into a routine if a state change is required.
- well-defined expression**— A term or group of terms that contains only symbols or assembly-time constants that have been defined before they appear in the expression.
- word**— A 16-bit addressable location in target memory

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products

| | |
|-----------------------------|--|
| Amplifiers | amplifier.ti.com |
| Data Converters | dataconverter.ti.com |
| DLP® Products | www.dlp.com |
| DSP | dsp.ti.com |
| Clocks and Timers | www.ti.com/clocks |
| Interface | interface.ti.com |
| Logic | logic.ti.com |
| Power Mgmt | power.ti.com |
| Microcontrollers | microcontroller.ti.com |
| RFID | www.ti-rfid.com |
| RF/IF and ZigBee® Solutions | www.ti.com/lprf |

Applications

| | |
|--------------------|--|
| Audio | www.ti.com/audio |
| Automotive | www.ti.com/automotive |
| Broadband | www.ti.com/broadband |
| Digital Control | www.ti.com/digitalcontrol |
| Medical | www.ti.com/medical |
| Military | www.ti.com/military |
| Optical Networking | www.ti.com/opticalnetwork |
| Security | www.ti.com/security |
| Telephony | www.ti.com/telephony |
| Video & Imaging | www.ti.com/video |
| Wireless | www.ti.com/wireless |

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2009, Texas Instruments Incorporated