Texas Instruments

MSP430 Family

Assembly Language Tools User's Guide

1994

User's Guide

MSP430 Family Assembler

1994

MSP430 Family Assembly Language Tools User's Guide

Read This First

This preface summarizes the chapters, lists related documentation, and describes the style and symbol conventions used in this manual.

How This Manual Is Organized

This document contains the following chapters:

Chapter 1	**Introduction and Installation** Provides an overview of the assembly language development tools, a walkthrough, and installation information.
Chapter 2	**Introduction to Common Object File Format** Discusses the basic COFF concept of **sections** and how they can help you use the assembler and linker more efficiently. Common object file format, or COFF, is the object file format used by the MSP430 family tools. *Read Chapter before using the assembler and linker.*
Chapter 3	**Assembler Description** Tells you how to invoke the assembler and discusses source statement format, valid constants and expressions, and assembler output.
Chapter 4	**Assembler Directives** Divided into two parts: the first part describes the directives according to function, and the second part presents the directives in alphabetical order.
Chapter 5	**Instruction Set Summary** Summarizes the MSP430 instruction set alphabetically.
Chapter 6	**Macro Language** Describes macro directives, substitution symbols used as macro parameters, and how to create macros.
Chapter 7	**Archiver Description** Contains instructions for invoking the archiver, creating new archive libraries, and modifying existing libraries.
Chapter 8	**Linker Description** Tells you how to invoke the linker, provides details about linker operation, discusses linker directives, and presents a detailed linking example.
Chapter 9	**Absolute Lister Description** Tells you how to invoke the absolute lister so that you can obtain a listing of the absolute addresses of an object file.

Chapter 10	**Object Format Converter Description** Tells you how to invoke the object format converter so that you can convert a COFF object file into an Intel, Tektronix, or TI-tagged object format.
Appendix A	**Common Object File Format** Contains supplemental technical data about the internal format and structure of COFF object files.
Appendix B	**Symbolic Debugging Directives** Lists symbolic debugging directives that a high level language can use.
Appendix C	**Assembler Error Messages** Lists the assembler error messages.
Appendix D	**Linker Error Messages** Lists the linker error messages.
Appendix E	**ASCII Character Set** Provides a table of the ASCII character set.
Appendix F	**Glossary** Contains a glossary of terms and acronyms used in this book.
Appendix G	**Floating Point Formats** Contains informations about the internal format of floating point constants.

Related Documentation

The following MSP430 documents are also available.

The MSP430 Family Data Manual (literature number SPNSxxx) discusses hardware aspects of the MSP430, such as pin functions, architecture, stack operation, and interfaces, and contains the MSP430 instruction set.

The MSP430 data sheets contain the recommended operating conditions, electrical specifications, and timing characteristics of the MSP430 family devices.

- ***MSP430C201 16–Bit Microcontroller Data Sheet*** (literature number SPNSxxx)

Style and Symbol Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a special font. Examples use a bold version of the special font for emphasis. Here is a sample program listing:

  ```
  1 0000  2δ      ξ   .βψτε  45
  2 0001  2φ      ψ   .βψτε  47
  3 0002  32      ζ   .βψτε  50
  4 0003              .τεξτ
  ```

- In syntax descriptions, the instruction, command, or directive is in a **bold face font** and parameters are in *italics*. Portions of a syntax that are in **bold face** should be entered as shown; portions of a syntax that are in *italics* describe the type of information that should be entered. Here is an example of a directive syntax:

  .space *size*

  .space is the directive. This directive has one parameter, indicated by *size*. When you use .space, the first and only parameter must be the size.

- Square brackets (**[** and **]**) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you don't enter the brackets themselves. This is an example of an instruction that has an optional parameter:

  .text *[address]*

- Braces ({ and }) indicate a list. The symbol I (read as *or*) separates items within the list. Here's an example of a list:

  `{ a | b | c }`

  This provides three choices: a, b, or c.

- Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. The syntax for this directive is:

  .byte *value*$_1$ [, ... , *value*$_n$]

  This syntax shows that .byte must have at least one value parameter, but you have the option of supplying additional value parameters, separated by commas.

Following are other symbols and abbreviations used throughout this document.

Symbol	Definition	Symbol	Definition
R0-R3	Registers with special functions	R4-R15	Working registers, general purpose
PC	Program counter register	SP	Stack pointer register
SR	Status register	CG1,CG2	Constant generator registers
LSB	Least significant bit	MSB	Most significant bit
H,h	Suffix – hexadecimal number	B,b	Suffix – binary integer
Q,q	Suffix – octal integer		
{ }	List of parameters	[ ]	Optional parameter
text	Indicates a “fill in the blank" – replace the text in *italics* with an appropriate substitute. For example, substitute an actual label for *label*; substitute an actual destination expression for *expression*.		

Trademarks

IBM, IBM PC, IBM PC/XT, PC–DOS, IBM OS/2, and PS/2 are trademarks of International Business Machines Corp.

MS, MS OS/2, MS–DOS, and MS–Windows are registered trademarks of Microsoft Corp.

Sun–3, Sun–4, SunView, SunWindows, and Sun Workstation are trademarks of Sun Microsystems, Inc.

UNIX is a registered trademark of AT&T Bell Laboratories, Inc.

VAX and VMS are trademarks of Digital Equipment Corp.

Topics

Figures

1 Introduction and Installation

The MSP430 devices are well supported by a full set of hardware and software development tools. This document discusses the software development tools included with the MSP430 assembly language package:

- Assembler
- Archiver
- Linker
- Absolute Lister
- ROM Utility

These tools can be installed on the following systems:

- PC/AT with PC–DOS, MS–DOS, OS/2 or MS-Windows

The MSP430 assembly language tools create and use object files that are in common object file format (COFF) to facilitate modular programming. Object files contain separate blocks (called sections) of code and data that you can load into different MSP430 memory spaces. You will be able to program the MSP430 more efficiently if you have a basic understanding of COFF.

1.1 Development Tools Overview

The figure shows the assembly language development flow. The shaded portion highlights the most common development path; the other portions are optional.

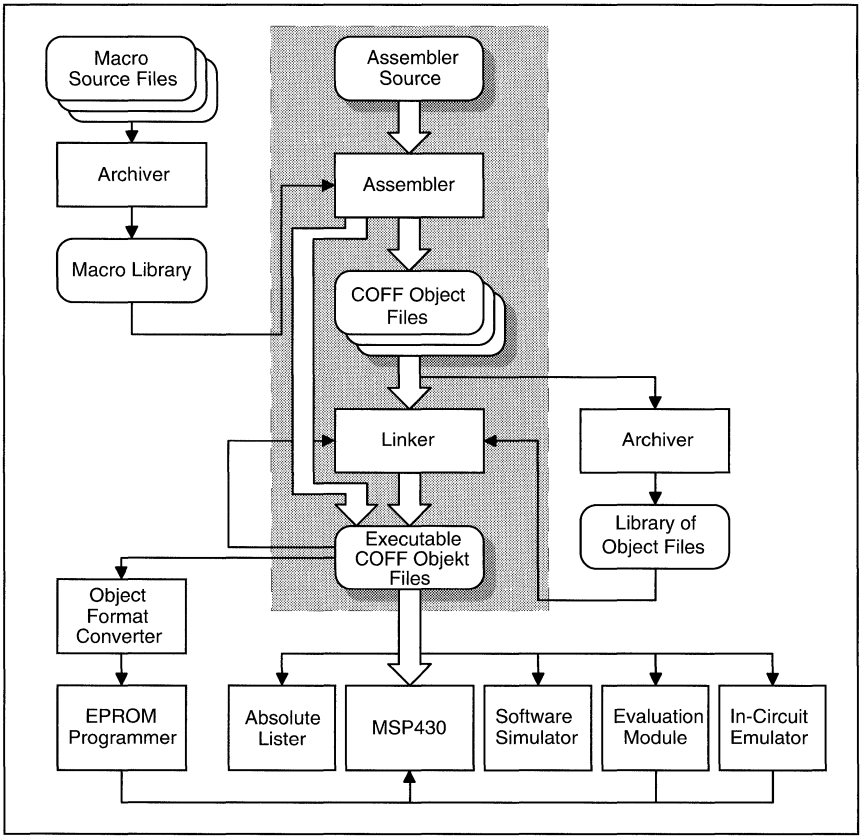


Figure 1.1: MSP430 Assembly Language Development Flow

- The **assembler** translates assembly language source files into machine language object files. 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, symbol definition, and section content.

- The **archiver** allows you to collect a group of files into a single archive file. For example, you can collect several macros together into a macro library. The assembler will search through the library and use only 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 will include in the library the members that resolve external references during the link.

- 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 COFF object files (created by the assembler) as input. It also accepts archiver library members and output modules created by a previous linker run. Linker directives allow you to combine object file sections, bind sections or symbols to addresses or within memory ranges, and define or redefine global symbols.

- The **absolute lister** provides a file that can be reassembled to produce a listing of the absolute addresses of an object file.

- The MSP430 microcontroller programmer accepts COFF files as input, but most EPROM programmers do not. The **object format converter** converts a COFF object file into TI–tagged, Intel, Motorola or Tektronix object format. The converted file can be downloaded to an EPROM programmer.

- The main purpose of this development process is to produce a module that can be executed in a system that contains a MSP430 device. You can use one of several debugging tools to refine and correct your code before downloading it to a MSP430 system.

1.2 Software Installation

This section contains instructions for installing the assembly language tools.

1.2.1 Installing the Tools on IBM PC/ATs or 100% Compatible Machines With PC–DOS, MS–DOS, OS/2 or MS-Windows

The MSP430 assembly language software package is shipped on a double–sided, high–density disk. Your system must have at least 512K bytes of memory space and 1MB of harddisk space.

First make a backup of the product disk.
Insert the backup disk into the floppy disk drive of your choice.
Change to that drive and enter:

```
INSTALL
```

Follow the instructions displayed on screen.

1.3 Getting Started

The tools you will probably use most often are the assembler and the linker. This section provides a quick walkthrough so that you can get started without reading the whole user's guide. These examples show the most common methods for invoking the assembler and linker.

1) Create two short source files to use for the walkthrough; call them `file1.asm` and `file2.asm`.

file1.asm	file2.asm

```
        .global    addq
start   clr R10
        clr R11


loop    call #addq
        jnc  loop
        .end
```
	```
	        .global    addq
	addq    inc        R10
	        inc        R11

	skp     ret
	        .end
	```

2) Enter the following command to assemble file1.asm.

```
asm430 file1
```

3) The **asm430** command invokes the MSP430 assembler; `file1.asm` is the input source file.

   If the input file extension is .asm, you don't have to specify the extension; the assembler uses .asm as the default. This example creates an object file called `file1.obj`. The assembler creates an object file only if there are no errors. You can specify a name for the object file, but if you don't, the assembler will append the .obj extension to the input filename.

4) Now assemble `file2.asm`; enter:

```
asm430 file2 -l
```

5) This time, the assembler creates an object file called `file2.obj`. The -l (lowercase "L") option tells the assembler to create a listing file; the listing file for this example is called `file2.lst`.

6) Link `file1.obj` and `file2.obj`; enter:

```
lnk430 file1 file2 -o prog.out
```

7) The **lnk430** command invokes the linker. `file1.obj` and `file2.obj` are the input object files. (If the input file extension is .obj, you don't have to specify the extension; the linker uses .obj as the default.) The linker combines `file1.obj` and `file2.obj` to create an executable object module called `prog.out`. The -o option supplies the name of the output module.

Topics

Figures

Notes

2 Introduction to Common Object File Format

The assembler and linker create object files that can be executed by a MSP430 device. The format that these object files are in is called *common object file format* (COFF).

COFF makes modular programming easier because it encourages 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.

For more information about COFF object file structure refer to the Appendix.

2.1 Sections

The smallest unit of an object file is called a **section**. A section is a block of code or data that will ultimately occupy contiguous space in the MSP430 memory map. Each section of an object file is separate and distinct from the other sections. COFF object files always 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.

It is important to note that 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, reg, and .regpair assembler directive are also uninitialized.

The assembler provides several directives that 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 that is organized like the object file shown in the following figure.

One of the linker's functions is to relocate sections into the target memory map; this is called **allocation**. Because most systems contain several different types of memory, using sections can help you to use target memory more efficiently. All sections are independently relocatable; you can place different sections into various blocks 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.

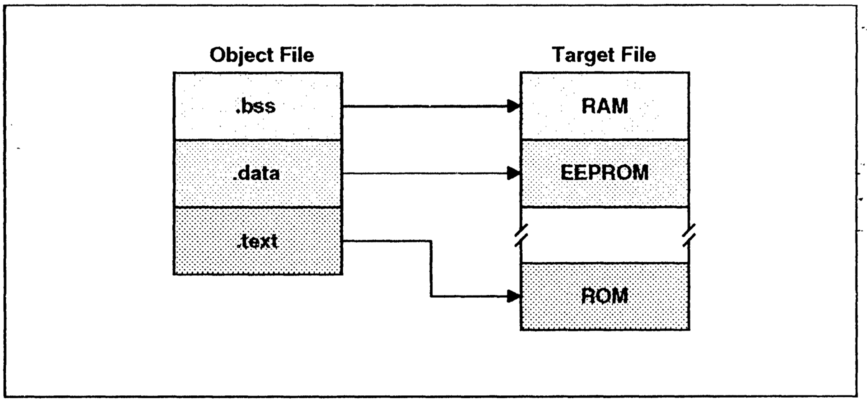


Figure 2.1: Partitioning Memory Into Logical Blocks

2.2 How the Assembler Handles Sections

The assembler's main function related to sections is to identify the portions of an assembly language program that belong in a particular section. The assembler has seven directives that support this function:

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

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

> **Note: Default Section Directive**
> If you don´t 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 runtime 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. If the section name is specified, the space is reserved in the named section. Each time you invoke one of these directives, the assembler reserves more space in the appropriate section.

The syntaxes for these directives are:

.bss *name* [*,size in bytes*]

symbol **.usect** *"section name", size in byte*

symbol	points to the first byte reserved by this invocation of the .bss or .usect directive. The symbol corresponds to the name of the variable that you're reserving space for. It can be referenced by any other section and can also be declared as a global symbol (with the .global assembler directive).
size	is an absolute expression. The .bss directive reserves size bytes in the .bss section; the .usect directive reserves size bytes in section name. If the section name is specified, the space is reserved in the named section.The default size for .bss is one byte.

section name tells the assembler which named section to reserve space in.

The .text, .data, and .sect directives tell the assembler to stop assembling into the current section and begin assembling into the indicated section. The .bss and .usect, 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 the contents of the initialized section.

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 separately 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:

.text

.data

.sect *“section name"*

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

Sections are built up 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 that are already in the .data section. This creates a single .data section that can be allocated contiguously into memory.

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 from the default sections.

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 don't want allocated with .text. If you assemble this segment of code into a named section, it will be assembled separately from .text, and you will be able to allocate it into memory separately from .text. Note that 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 sections that are used like the .bss section. These sections reserve space in RAM for variables.
- The **.sect** directive creates sections that can contain code or data, similar to the default .text and .data sections. The .sect directive creates named sections with *relocatable* addresses.

The syntaxes for these directives are:

symbol **.usect** *"section name", size*

.sect *"section name"*

The *section name* parameter is the name of the section. Section names are significant to 8 characters. You can create up to 32,767 separate named sections.

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 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 at that point.

The assembler treats each section as if it begins at address 0; the linker relocates each section according to its final location in the memory map.

2.2.5 An Example That Uses Sections Directives

The figure on the next page shows how you can build COFF 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.
- Continue assembling into a section that already contains code. In this case, the assembler simply appends the new code to the code that is already in the section.

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.

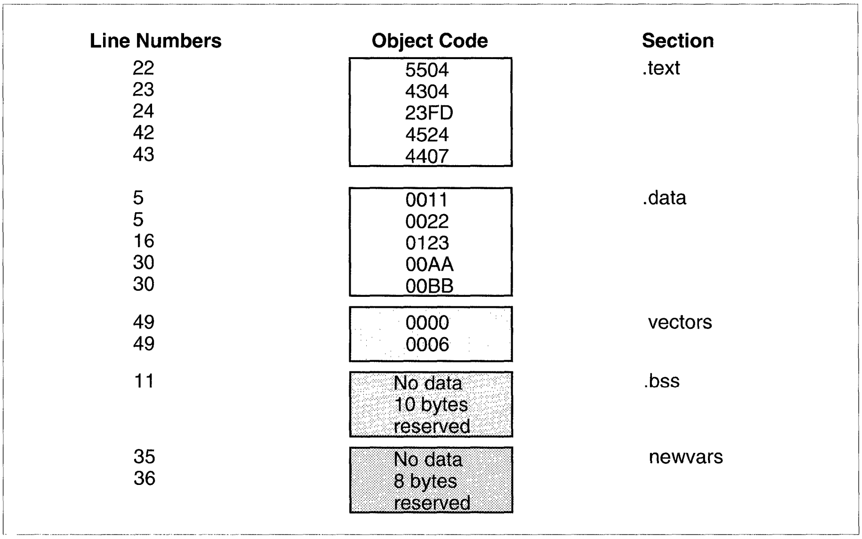
```
 1                    *********************************************
 2                    ** Assemble an initialized table into .data **
 3                    *********************************************
 4 0000                        .data
 5 0000   0011        coeff    .word  011h, 022h
   0002   0022
 6
 7                    *********************************************
 8                    ** Reserve space in .bss for a variable    **
 9                    *********************************************
10
11 0000                        .bss  buffer, 10
12                    *********************************************
13                    ** Still in .data                          **
14                    *********************************************
15
16 0004   0123        ptr      .word  0123h
17                    *********************************************
18                    ** Assemble code into the .text section    **
19                    *********************************************
20
21 0000                        .text
22 0000   5504        addl     add R5,R4
23 0002   4304                 clr R4
24 0004  '23fd                 jnz addl
25                    *********************************************
26                    ** Assemble more data into the .data section**
27                    *********************************************
28
29 0006                        .data
30 0006   00aa        ivals    .word  0aah, 0bbh
   0008   00bb
31                    *********************************************
32                    ** define another section for more variables**
33                    *********************************************
34
35 0000               var2     .usect "newvars",1
36 0001               inbuf    .usect "newvars",7
37                    *********************************************
38                    ** Assembler more code into .text          **
39                    *********************************************
40
41 0006                        .text
42 0006   4524        acode    mov @R5, R4
43 0008   4407                 mov R4, R7
44                    *********************************************
45                    ** Define a named section for int. vectors **
46                    *********************************************
47
48 0000                        .sect  "vectors"
49 0000  '0000                 .word  addl, acode
   0002  '0006

Field 1 Field 2 Field 3                Field 4
```

Figure 2.2: Using Sections Directives

The listing file creates five sections:

.text	contains 10 bytes of object code.
.data	contains 10 bytes of object code.
vectors	is a named section created with the .sect directive; it contains 4 bytes of initialized data.
.bss	reserves 10 bytes in memory.
newvars	is a named section created with the .usect directive; it reserves 8 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.



Line Numbers	Object Code	Section
22	5504	.text
23	4304	
24	23FD	
42	4524	
43	4407	
5	0011	.data
5	0022	
16	0123	
30	00AA	
30	00BB	
49	0000	vectors
49	0006	
11	No data 10 bytes reserved	.bss
35	No data 8 bytes reserved	newvars
36		

Figure 2.3: Generated Object Code according to previous source code example

2.3 How the Linker Handles Sections

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

The linker provides two directives that 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 and where to place the output sections in memory.

It is not always necessary to use linker directives. If you don't use them, the linker uses the target processor's default allocation algorithm. When you *do* use linker directives, you must specify them in a linker command file.

2.3.1 Default Memory Allocation

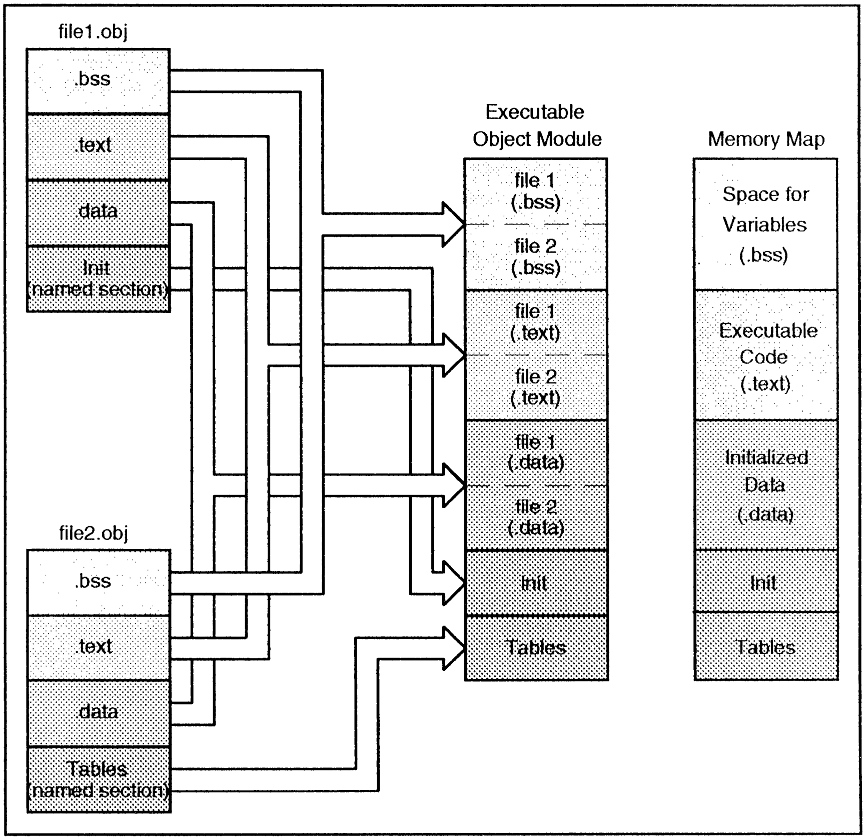


Figure 2.4: Combining Input Sections to Form an Executable Object Module

In the figure, 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 output module shows the combined sections. The linker combines file1.text with file2.text to form one .text section, then combines the .data sections, then the .bss sections, and finally places the named sections at the end. The memory map shows how the sections are put into memory.

2.3.2 Placing Sections in the Memory Map

The figure also illustrates the linker's default methods 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 might want a named section placed where the .data section would normally be allocated. Most memory maps comprise various types of memories (RAM, ROM, EPROM, etc.) in varying amounts; you may want to place a section in a particular type of memory.

- The **MEMORY** directive allows you to define the memory map for your particular system.
- The **SECTIONS** directive lets you build sections and place them into memory.

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 can't actually begin at address 0 in memory, so the linker **relocates** sections by:

- allocating sections into the memory map so that they begin at the appropriate address.
- 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.

```
                                 .global x
0000                             .text
0000   !40300000                 br   #x     ;uses an external relocation
0004   '12B00008                 call #y     ;uses an internal relocation
0008    5504            y:       add  R5, R4 ;defines internal relocation
```

Figure 2.5: An Example of Code That Generates Relocation Entries

Both symbols x and y are relocatable. y is defined in the .text section of this module; x is defined in some other 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 (indicated by the **!** character in the listing). The reference to y is to an internally defined relocatable symbol (indicated by the ' character in the listing).

After the code is linked, suppose that x is relocated to address 7100h. Suppose also that the .text section is relocated to begin at address 7200h; y now has a relocated value of 7208h. The linker uses the two relocation entries to patch the two references in the object code:

```
40300000  br    #x  becomes 40307100
12B00008  call  #y  becomes 12B07208
```

Each section in a COFF object file 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 -r option.

2.5 Runtime Relocation

It may be necessary or desirable at times to load code into one area of memory and run it in another. For example, you may have performance–critical code in a ROM–based system. The code must be loaded into ROM but would run much faster if it were in RAM.

The linker provides a simple way to specify this. In the SECTIONS directive, you can optionally direct the linker to allocate a section twice: once 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 will place 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 just because 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 will load and run at the same address. If you provide both allocations, the section is actually allocated as if it were two different 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.

2.6 Loading a Program

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

In order 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. Some of the more common situations are listed below.

- The MSP430 development tools (In-Circuit-Emulator and Evaluation Module) provide COFF object module loading capabilities.
- You can use the object format converter (the rom430, which is shipped as part of the assembly language package) to convert the executable COFF object module into one of several object file formats. You can then use the converted file with almost any EPROM programmer to burn the program into an EPROM.

2.7 Symbols in a COFF File

A COFF file contains a symbol table that stores information about symbols in the program. The linker uses this table when it performs relocation. Debugging tools can also use the symbol table to provide symbolic debugging.

2.7.1 External Symbols

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

Defined (.def)	Defined in the current module and used in another module
Referenced (.ref)	Referenced in the current module, but defined in another module
Global (.global)	May be either of the above

The following code segment illustrates these definitions.

```
x:    ADD         #56h, R4    ;     Define x
      BR          #y          ;     Reference y
      .global     x           ;     DEF of x
      .global     y           ;     REF of y
```

The .global definition of x says that it is an external symbol defined in this module and that other modules can reference x. The .global definition of y says that it is an undefined symbol that is defined in some other 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 defines unresolved references to x from 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). 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 in a section.

The assembler does not usually create symbol table entries for any other type of symbol, because the linker does not use them. For example, labels are not included in the symbol table unless they are declared with .global. For symbolic debugging 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 -s option.

Topics

3 Assembler Description

The assembler translates assembly language source files into machine language object files. These files are in common object file format (COFF). Source files can contain the following assembly language elements:

Assembler directives

Assembly language instructions

Macro directives

This two–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 an SPC (section program counter) for each section of object code.
- Defines and references global symbols and appends a cross–reference listing to the source listing (if requested).
- Assembles conditional blocks.
- Supports macros, allowing you to define macros inline or in a library.

3.1 Assembler Development Flow

The figure illustrates the assembler's role in the assembly language development flow. The assembler accepts assembly language source files as input.

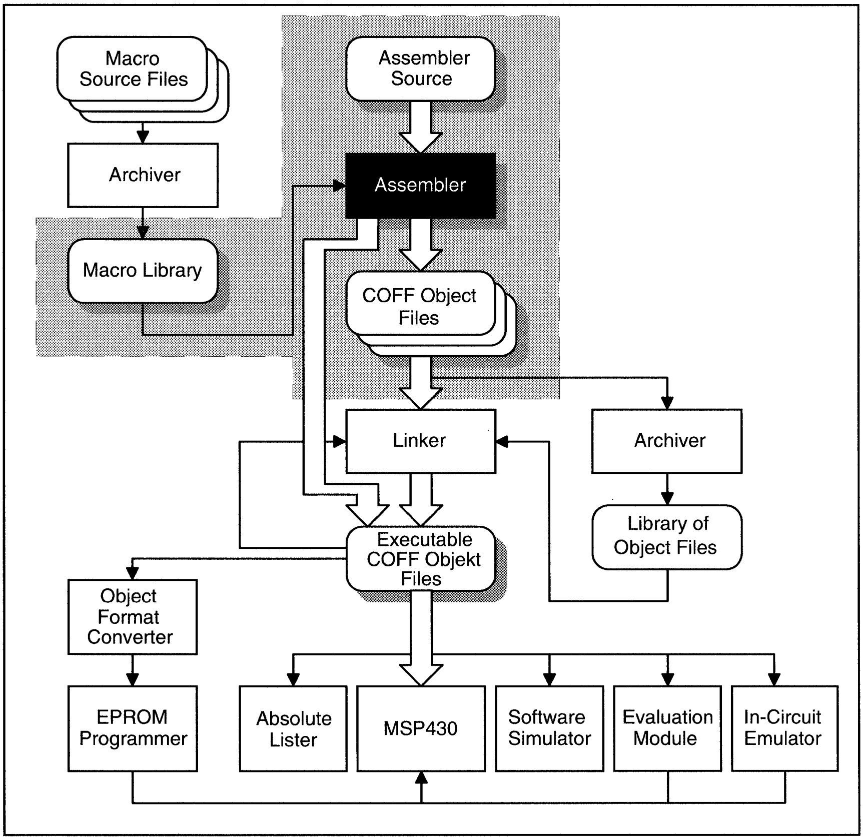


Figure 3.1: Assembler Development Flow

3.2 Invoking the Assembler

To invoke the assembler, enter the following:

```
asm430  [input file  [object file  [listing file  [ rawdata file]]]]  [-options]
```

asm430 is the command that invokes the assembler.

input file names the assembly language source file. If you do not supply an extension, the assembler assumes that the input file has the default extension *.asm.* If you do not supply an input filename when you invoke the assembler, the assembler will prompt you for one.

object file names the object file that the assembler creates. If you do not supply an extension, the assembler uses *.obj* as a default extension. If you do not supply an object file, the assembler creates a file that uses the input file name with the *.obj* extension.

listing file names the optional listing file that the assembler can create. If you do not supply a name for a listing file, *the assembler does not create one* unless you use the -l option. In this case, the assembler uses the input filename. If you do not supply an extension, the assembler uses *.lst* as a default extension.

rawdata file names the optional ascii-format object file that the assembler can create. If you do not supply a name for the rawdata file, the assembler does not create one unless you use the -z option. In this case, the assembler uses the input filename. If you do not supply an extension, the assembler uses .txt as a default extension

options identifies 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 (-). You can string the options together; for example, **-lc** is equivalent to **-l -c**. The valid assembler options are as follows:

-a creates an absolute listing. When you use -a, the assembler does not produce an object file. The absolute listing option is used in conjunction with the absolute lister.

-b suppress banner on all pages except page 1.

-c makes case insignificant. For example, the symbols *ABC* and *abc* will be equivalent. *If you do not use this option, case is significant.*

-i specifies a directory where the assembler can find files named by the .copy, .include, or .mlib directives. The format of the -i option is **-i***pathname*. You can specify up to 10 directories in this manner; each pathname must be preceded by the -i option.

-l (lowercase “L") produces a listing file.

-q (quiet) suppresses the banner and all progress information.

-s puts **all** defined symbols in the object file's symbol table. Usually, the assembler puts only global symbols into the symbol table. When you use -s, symbols that are defined as labels or as assembly–time constants are also placed in the symbol table.

-x produces a cross–reference table and appends it to the end of the listing file. If you do not request a listing file, the assembler creates one anyway.

-z creates an objext file in ascii format containing no relocation and debug information. The generation of the COFF object file is not affected. This option is used in conjunction with the evaluation module.

3.3 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. The syntaxes for these directives are:

.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. The *filename* may be a complete pathname, a partial pathname, or a filename with no path information. The assembler searches for the file in:

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 -i assembler option.
3) Any directories set with the environment variable A_DIR.

You can augment the assembler's directory search algorithm by using the -i assembler option or the A_DIR environment variable.

3.3.1 -i Assembler Option

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

asm430 -i*pathname source filename*

You can use up to 10 -i options per invocation; each -i option names one *pathname*. In assembly source, you can use the .copy, .include, or .mlib directive without specifying any path information. If the assembler doesn't find the file in the directory that contains the current source file, it searches the paths provided by the -i 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"
```

	Pathname for `copy.asm`	Invocation Command
DOS	`c:\430\files\copy.asm`	`asm430 -ic:\430\files source.asm`

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 -i option.

3.3.2 Environment Variable (A_DIR)

An environment variable is a system symbol that you define and assign a string to. The assembler uses the environment variable **A_DIR** to name alternate directories that contain copy/include files or macro libraries. The command for assigning the environment variable is as follows:

DOS `set A_DIR =` *pathname;another pathname ...*

The *pathnames* are directories that contain copy/include files or macro libraries. You can separate the pathnames with a semicolon or with blanks. In assembly source, you can use the .copy, .include, or .mlib directive without specifying any path information. If the assembler doesn't find the file in the directory that contains the current source file or in directories named by -i, 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"
```

	Pathname	Invocation Command

DOS	`c:\430\files\copy1.asm`
	`c:\dsys\copy2.asm`
		`set A_DIR=c:\dsys; c:\exec\source`
		`asm430 -ic:\430\files source.asm`

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 -i option and finds copy1.asm. Finally, the assembler searches the directory named with A_DIR and finds copy2.asm.

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

DOS `set A_DIR=`

3.4 Source Statement Format

MSP430 assembly language source programs consist of source statements that can contain assembler directives, assembly language instructions, macro directives, and comments. Source statement lines can be as long as the source file format allows, but the assembler reads up to 200 characters per line. If a statement contains more than 200 characters, the assembler truncates the line and issues a warning.

The next several lines show examples of source statements:

```
sym      .equu   2              ; Symbol sym = 2
Begin:   ADD     #sym+5,R11     ; Add (sym+5) to contents of R11
         .word   016h           ; Initialize a word with 016h
```

A source statement can contain four ordered fields. The general syntax for source statements is as follows:

[label] [:]   *mnemonic*   *[operand list]*   *[;comment]*

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. Note that tab characters are equivalent to blanks.
- 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.

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 32 alphanumeric characters (A-Z, a-z, 0-9, _, and $). Labels are case–sensitive, 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 don't 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 section program counter (the label points to the statement it's associated with). If, for example, you use the .word directive to initialize several words, a label would point to the first word. In the following example, the label Start has the value 40h.

```
.        .        .        .
.        .        .        .
.        .        .        .
9        003F              * Assume some other code was assembled
10       0040     000A     Start: .word 0Ah,3,7
         0041     0003
         0043     0007
```

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

```
label    .equu    $  ;  $ 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
```

Mnemonic Field

The mnemonic field follows the label field. *The mnemonic field cannot start in column 1, or it would be interpreted as a label.* The mnemonic field can contain one of the following opcodes:

- Machine–instruction mnemonic (such as ADC, MOV, POP)
- Assembler directive (such as .data, .list, .equ)
- Macro directive (such as .macro, .var, .mexit)
- A macro call

Operand Field

The operand field is a list of operands that follow the mnemonic field. An operand can be a constant, a symbol, or a combination of constants and symbols in an expression. You must separate operands with commas.

- **Operand Prefixes for Instructions**

  The assembler allows you to specify that a constant, symbol, or expression should be used as an address, an immediate value, or an indirect value. The following rules apply to the operands of instructions.

  - **No prefix — the operand is an address or a register**. If you do not use a prefix with an operand, the assembler treats an operand representing a constant value as an absolute address. When the operand is a label, the assembler generates a symbolic address. A register name specifies the contents of the named register. This are examples of instructions that use operands without prefixes:

    ```
    Label: ADD 0FFFEh,R5   ; add contents of absolute address to register
           ADD Label,R5    ; add contents of symbolic address to register
    ```

  - **& prefix — the operand is an absolute address**. If you use the **&** sign as a prefix, the assembler treats the operand as an absolute address, similar to using no prefix. The operand has to specify a constant value:

    ```
           MOV &200h,R5
           MOV 200h,R5
    ```

    Both instructions generate the same object code, moving the contents of absolute address 200h to register R5.

  - **# prefix — the operand is an immediate value**. If you use the # sign as a prefix, the assembler treats the operand as an immediate value. This is true even when the operand is an address; the assembler treats the address as a value instead of using the contents of the address. This is an example of an instruction that uses an operand with the # prefix:

    ```
    Label: ADD #123,R5
    ```

    The operand #123 is an immediate value. The assembler adds 123 (decimal) to the contents of register R5.

  - **@ prefix — the operand is an indirect address.** If you use the @ sign as a prefix, the assembler treats the operand as an indirect address; that is, it uses the contents of the operand as an address. This is an example of an instruction that uses an operand with the @ prefix:

    ```
    Label: MOV @R4,R4
    ```

    The operand @R4 specifies an indirect address. The assembler goes to the address specified by the contents of register R4 and then moves the contents of that location to register R4.

- **Immediate Addressing for Directives**

  The immediate addressing mode is used mostly with instructions; in some cases, it can also be used with the operands of directives.

  Usually, it is not necessary to use the immediate addressing mode for directives. Compare the following statements:

  ```
  ADD #10, R4
  .byte 10
  ```

  In the first statement, the immediate addressing mode is necessary to tell the assembler to add the value 10 to register R4. In the second statement, however, immediate addressing is not used; the assembler expects the operand to be a value and initializes a byte with the value 10.

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.

3.5 Constants

The assembler supports six types of constants:

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

The assembler maintains each constant internally as a 32–bit quantity. Note that constants **are not sign extended**. For example, the constant 0FFH is equal to 00FF (base 16) or 255 (base 10); it **does not** equal -1.

Binary Integers

A binary integer constant is a string of up to 16 binary digits (0s and 1s) followed by the suffix **B** (or **b**). If fewer than 16 digits are specified, the assembler right–justifies the value and zero–fills the unspecified bits. 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 $0F8_{16}$**

Octal Integers

An octal integer constant is a string of up to 6 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}**
100000Q	**Constant equal to $32{,}768_{10}$ or 8000_{16}**
226Q	**Constant equal to 150_{10} or 96_{16}**

Decimal Integers

A decimal integer constant is a string of decimal digits, ranging from? -32,768 to 65,535. 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}**

Hexadecimal Integers

A hexadecimal integer constant is a string of up to 4 hexadecimal digits followed by the suffix **H** (or **h**). 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 4 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}**
0Fh	**Constant equal to 15_{10} or $000F_{16}$**
37ACh	**Constant equal to $14{,}252_{10}$ or $37AC_{16}$**

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 *a* and is represented internally as 61_{16}**
'C'	**Defines the character constant *C* and is represented internally as 43_{16}**
''''	**Defines the character constant *''* and is represented internally as 27_{16}**
''	**Defines a null character and is represented internally as 00_{16}**

Note the difference between character *constants* and character *strings.* A character constant represents a single integer value; a string is a list of characters.

Assembly-Time Constants

If you use the .equ directive to assign a value to a symbol, the symbol becomes a constant. In order to use this constant in expressions, the value that is assigned to it must be absolute. For example:

```
sym      .equ   3
         MOV #sym,R10
```

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

```
sym      .equ   R14
         MOV #10,sym
```

3.6 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. Appendix lists valid characters.

These are examples of valid character strings:

"sample program"	**defines a 14–character string, sample program**
"PLAN ""C"""	**defines an 8–character string, PLAN "C"**

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"
- Operand of .string or .byte directive

3.7 Symbols

Symbols are used as labels, constants, and substitution symbols. A symbol name is a string of up to 32 alphanumeric characters (A-Z, a-z, 0-9, $, _and?). The first character in a symbol cannot be a number; 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 -c assembler option. This type of symbol is valid only during the assembly in which it is defined, unless you use the .global directive to declare it as an external symbol.

Labels

Symbols that are used as labels become symbolic addresses that are associated with locations in the program. A label used locally within a file must be unique. Mnemonic op-codes and assembler directive names (without the _`.' prefix) are valid label names.

Labels can also be used as the operand of a .global, .ref, .def, or .bss directive; for example:

```
        .global   label1

label2  nop
        mov       label1, R4
        br        label2
```

Local Labels

Local labels are a special type of label whose scope and effect are only temporary. A local label has the form $*n*, where *n* is a decimal digit in the range 0-9. For example, $4 and $1 are valid local labels.

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. If a local label is used as an operand, it can be used only as an operand for a 10–bit jump instruction.

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

- By 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)

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

```
Label1:     mov R12,R13
            jnz $1
            mov #-1,R13
$1          cmp R13,R4
            .newblock; Undefine $1 so it can be used again
            jne $1
            inc R13
$1          add R13,R14
```

The following code uses a local label illegally:

```
Label1:     mov R12,R13
            jnz $1
            mov #-1,R13
$1          cmp R3,R4
            jne $1
            inc R13
$1          add R13,R14  ; WRONG – $1 is multiply defined
```

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. However, if you use a local label within a macro and then use .newblock within the macro, the local label is used and reset each time the macro is expanded.

Up to ten local labels can be in effect at one time. 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.

Constants

Symbols can be set to constant values. By using constants, you can equate meaningful names with constant values. The .equ, .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
         .byte  value           ; constant offsets value = 0
         .byte  delta           ; constant offsets delta = 1
i_len    .endstruct

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

         MOV    array.delta,R4  ; array+1
```

The assembler also has several predefined symbolic constants; these are discussed in the next subsection.

Symbolic Constants

The assembler has several predefined symbols, including the following:

- **$**, the dollar sign character, represents the current value of the section program counter (SPC).
- **Register symbols**, which are of the form R*n* or r*n* , where *n* is an expression that evaluates in the range 0-15. (If the number is greater than 15, the symbol is not considered a register symbol.) The number may be decimal. Note that **PC**, **SP** and **SR** are valid register symbols; they represent registers with special functions (R0 - R3).

Substitutions 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    "R13",   SP1
        .asg    "+",     pls
        .asg    "-5",    min5

        ADD     # min5,SP1
```

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

```
add2    .macro    src,dest    ;  add2 macro definition

        mov     src,R4
        mov     R4,R5
        mov     dest,R4
        add     R5,R4
        mov     R4,dest

        .endm

*add2 invocation
        add2    loc1, loc2
```

3.8 Expressions

An expression is a constant, a symbol, or a series of constants and symbols separated by arithmetic operators. The range of valid expression values is -32,768 to 65,535. These are the three main factors that influence the order of expression evaluation:

Parentheses Expressions that are enclosed in parentheses are always evaluated first.

8/(4/2) = 4, but 8/4/2 = 1

Note that you **cannot** substitute braces ({ }) or brackets ([ ]) for parentheses.

Precedence groups Operators, listed in the next table, are divided into nine precedence groups. When the order of expression evaluation is not determined by parentheses, 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. Note that the highest precedence group is evaluated from right to left.

8/4*2 = 4, but 8/(4*2) = 1

3.8.1 Operators

Group	Operator	Description
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 XOR
9	|	Bitwise OR

Notes: 1) Operators in parentheses ( ) indicate an alternate form.
2) Group 1 operators are evaluated right to left. All other operators are evaluated left to right.

Table 3.1: Operators Used in Expressions (Precedence)

3.8.2 Expression Overflow and Underflow

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

3.8.3 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.

3.8.4 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	= =	Equal to
! =	Not 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 may be used only on operands of equivalent types, e.g., absolute value compared to absolute value, but not absolute value compared to relocatable value.

3.8.5 Relocatable Symbols and Legal Expressions

The following table summarizes valid operations on absolute, relocatable, and external symbols. An expression cannot multiply or divide by a relocatable or external symbol. An expression cannot contain unresolved symbols that are relocatable with respect to different sections.

Symbols that have been defined as global with the .global directive can also be used in expressions; in the table, these symbols are referred to as *external.*

If A is...	If B is...	A + B is...	A - B is...
absolute	absolute	absolute	absolute
absolute	relocatable	relocatable	illegal
absolute	external	external	illegal
relocatable	absolute	relocatable	relocatable
relocatable	relocatable	illegal	absolute *
relocatable	external	illegal	illegal
external	absolute	external	external
external	relocatable	illegal	illegal
external	external	illegal	illegal

* A and B must be in the same section; otherwise, this is illegal.

Table 3.2: Expressions With Absolute and Relocatable Symbols

Here are some examples of expressions that use relocatable and absolute symbols. These examples use four symbols that are defined in the same section:

```
             .global extern_1   ;Defined in an external module
intern_1:    .word 'D'          ;Relocatable, defined in current module
LAB1:        .equ 2             ;LAB1 = 2
intern_2:                       ;Relocatable, defined in current module
```

- **Example 1**:

  The first statement in this example puts the value 51 into register R4. The second statement puts the value 27 into register R4.

  ```
  MOV        #(LAB1 + (4+3) * 7), R4        ; R4 = 51
  MOV        #(LAB1 + 4 + 3 * 7), R4        ; R4 = 27
  ```

- **Example 2**

  All legal expressions can be reduced to one of two forms:

  relocatable symbol ± absolute symbol

  or

  absolute value

  Unary operators can be applied only to absolute values; they cannot be applied to relocatable symbols. Expressions that cannot be reduced to contain only one relocatable symbol are illegal. The first statement in the following example is legal; the others are illegal.

  ```
  MOV  extern_1 - 10, R4      ; Legal
  MOV  10-extern_1, R4        ; Can't negate relocatable symbol
  MOV  -(intern_1), R4        ; Can't negate relocatable symbol
  MOV  extern_1/10, R4        ; / is not an additive operator
  MOV  intern_1 + extern_1,R4 ; Multiple relocatables
  ```

- **Example 3**

  The first statement below is legal; although intern_1 and intern_2 are relocatable, their difference is absolute because they're in the same section. Subtracting one relocatable symbol from another reduces the expression to *relocatable symbol + absolute value*. The second statement is illegal because the sum of two relocatable symbols is not an absolute value.

```
MOV        intern_1 - intern_2 + extern_1, R4   ; (legal)
MOV        intern_1 + intern_2 + extern_1, R4   ; (illegal)
```

- **Example 4**

  An external symbol's placement in an expression is important to expression evaluation. Although the statement below is similar to the first statement in the previous example, it is illegal. This is because of left–to–right operator precedence; the assembler attempts to add intern_1 to extern_1.

```
MOV        intern_1 + extern_1 - intern_2, R4   ; (illegal)
```

3.9 Source Listings

A source listing shows source statements and the object code they produce. To obtain a listing file, invoke the assembler with the -l (lowercase "L") option.

At the top of each source listing page are two banner lines, a blank line, and a title line. Any title supplied by a .title directive is printed on this line; a page number is printed to the right of the title. If you don't use the .title directive, the title area is left blank. The assembler inserts a blank line below the title line.

Each line in the source file may produce a line in the listing file that shows a source statement number, an SPC value, the object code assembled, and the source statement. A source statement may produce more than one byte of object code and may be listed on more than one line. If so, each additional line is listed immediately following the source statement line.

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 statements in the source file that are not listed.

Include File Letter

The assembler may precede a line with a letter; the letter indicates that the line is assembled from an include file.

Nesting Level Number

The assembler may precede a line with a number; the number indicates the nesting level of macro expansions and loop blocks.

Field 2 **Section Program Counter**

This field contains the section program counter (SPC) value (hexadecimal). All sections (.text, .data, .bss, and named sections) maintain separate SPCs. Some directives do not affect the SPC; they 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 contains two columns immediately preceding the object code, which indicate additional information about the line of object code.

The first column either will be blank or will contain an asterisk (*). The asterisk indicates that the object code is not a direct mapping from the assembly source.

The second column indicates a relocation type that is associated with one of the operands for this line of source code. If more than one operand is relocatable, this column indicates the relocation type for the first one. The characters that may appear in this column and their associated relocation types are illustrated in the following table:

!	external reference (global)	"	.data relocatable
'	text relocatable	-	.bss, .usect relocatable
+	.sect relocatable		

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.

Example of an assembler listing with each of the four fields identified:

```
       1                                 .global func3
       2 0000                            .bss data1,1
       3 0001                            .bss data2,1
       4
       5                                 .copy "mac1.inc"
       1                          add2   .macro src, dest
       2                                 mov src, R4
       3                                 mov R4, R5
       4                                 mov dest, R4
       5                                 add R5,R4
       6                                 mov R4, dest
       7                                 .endm
       8
       9                                ;******************************
      10                                ;**  interrupt vectors        **
      11                                ;******************************
      12 0000                           .sect "int_vecs"
      13 0000  '0000                    .word func1
      14 0002  +0000                    .word func2
      15 0004  !0000                    .word func3
      16
      17                                ;******************************
      18                                ;**  .text section            **
      19                                ;******************************
      20 0000                           .text
      21 0000                    func1:
      22 0000                           add2 data1, data2
1          0000  -40140000              mov data1, R4
1          0004   4405                  mov R4, R5
1          0006  -4014fff9              mov data2, R4
1          000a   5504                  add R5,R4
1          000c  -4480fff3              mov R4, data2
      33 0020  -40140000                mov  data1, R4
      34 0024   940a                    cmp R4,R10
      35 0026  '3401                    jge lab
      36 0028   4304                    clr R4
      37 002a   1300             lab:   reti
      38
      39 0000                           .sect "other_code"
      40 0000   9405             func2  cmp R4,R5
      41 0002   1300                    reti
```

Field 1   Field 2   Field 3   Field 4

Example 3.1: An Assembler Listing

3.10 Cross–Reference Listings

A cross–reference listing shows symbols and their definitions. To obtain a cross–reference listing, invoke the assembler with the -x option or use the .option directive. The assembler will append the cross–reference to the end of the source listing.

```
LABEL                               VALUE            DEFN         REF

data1                               0000    -           2          27  28
data2                               0001    -           3          27  27*
func1                               0000    ´          26          18
func2                               0000    +          40          19
func3                               REF                             1  20
lab                                 002a    ´          37          35
```

Example 3.2: An Assembler Cross–Reference Listing

LABEL column contains each symbol that was defined or referenced during the assembly.

VALUE column contains a 4–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. The next table 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. If the line number is followed by an asterisk (*), that reference may modify the contents of the object. A blank in this column indicates that the symbol was never used.

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

Table 3.3: Symbol Attributes

Topics

Examples

List of Tables

Notes

4 Assembler Directives

Assembler directives supply program data 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 that the assembler can obtain macros from
- Generate symbolic debugging information

4.1 Directives Summary

The table summarizes the assembler directives. *Note that all source statements that contain a directive may have a label and a comment.* To improve readability, they are not shown as part of the directive syntax.

Directives That Define Sections

Mnemonic and Syntax	*Description*
.bss symbol [, size in bytes, address]	Reserve size bytes in the .bss (uninitialized data) section
.data [*address*]	Assemble into the .data (initialized data) section
.sect *"section name"* [*, address*]	Assemble into a named (initialized) section
.text *[address]*	Assemble into the .text (executable code) section
symbol **.usect** *"section name", size in bytes [, address]*	Reserve *size* bytes in a named (uninitialized) section

Directives That Initialize Constants (Data and Memory)

Mnemonic and Syntax	*Description*
.byte *value*$_1$ [*, ... , value*$_n$]	Initialize one or more successive bytes in the current section
.double *floating point value*	Initialize a 48-bit MSP430 floating-point constant
.field *value* [*, size in bits*]	Initialize a variable–length field
.float *floating point value*	Initialize a 32-bit, MSP430 floating–point constant
.space *size in bytes*	Reserve *size* bytes in the current section; note that a label points to the beginning of the reserved space
.string *"string*$_1$*"* [*, ... , "string*$_n$*"*]	Initialize one or more text strings
.word *value*$_1$ [*, ... , value*$_n$]	Initialize one or more 16–bit integers

Table 4.1: Assembler Directives Summary

Directives That Align the Section Counter (SPC)	
Mnemonic and Syntax	*Description*
.align	Align the SPC on a byte boundary
.even	Align the SPC on a word boundary

Directives That Format the Output Listing	
Mnemonic and Syntax	*Description*
.fclist	Allow false conditional code block listing (default)
.fcnolist	Inhibit false conditional code block listing
.length *page length*	Set the page length of the source listing
.list	Restart the source listing
.mlist	Allow macro listings and loop blocks (default)
.mnolist	Inhibit macro listings and loop blocks
.nolist	Stop the source listing
.option *{A|B|F|M|T|W|X}*	Select output listing options
.page	Eject a page in the source listing
.sslist	Allow expanded substitution symbol listing
.ssnolist	Inhibit expanded substitution symbol listing (default)
.title *"string"*	Print a title in the listing page heading
.width *page width*	Set the page width of the source listing

Table 4.1: Assembler Directives Summary (Continued)

Directives That Reference Other Files

Mnemonic and Syntax	*Description*
.copy ["]*filename*["]	Include source statements from another file
.def *symbol*$_1$ [, ... , *symbol*$_n$]	Identify one or more symbols that are defined in the current module and used in other modules
.global *symbol*$_1$ [, ... , *symbol*$_n$]	Identify one or more global (external) symbols
.include ["]*filename*["]	Include source statements from another file
.mlib ["]*filename*["]	Define macro library
.ref *symbol*$_1$ [, ... , *symbol*$_n$]	Identify one or more symbols that are used in the current module but defined in another module

Conditional Assembly Directives

Mnemonic and Syntax	*Description*
.break [*well–defined expression*]	Optional repeatable block assembly
.if *well–defined expression*	Begin conditional assembly
.else	Optional conditional assembly
.elseif well–defined expression	Optional conditional assembly
.endif	End conditional assembly
.endloop	End repeatable block assembly
.loop [*well–defined expression*]	Begin repeatable block assembly

Table 4.1: Assembler Directives Summary (Continued)

Assembly–Time Symbols

Mnemonic and Syntax	*Description*
.asg ["] *character string* ["], *substitution*	Assign a character string to a substitution symbol
.endstruct	End structure definition
.equ	Equate a value with a symbol
.eval *well–defined expression, sub– stitution symbol*	Perform arithmetic on numeric substitution symbols
.newblock	Undefine local labels
.set	Equate a value with a symbol
.struct	Begin structure definition
.tag	Assign structure attributes to a label

Miscellaneous Directives

Mnemonic and Syntax	*Description*
.emsg string	Send user–defined error messages to the output device
.end	Program end
.label *"symbol"*	Define a load address label
.mmsg *string*	Send user–defined messages to the output device
.setsect *"section name",addr*	Produced by absolute lister, See Chapter 9
symbol **.setsym** *addr*	Produced by absolute lister, See Chapter 9
.wmsg *string*	Send user–defined warning messages to the output device

Table 4.1: Assembler Directives Summary (Concluded)

4.2 Directives That Define Sections

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

- **.bss** reserves space in the .bss section for uninitialized variables.
- **.data** identifies portions of code in the .data section. The .data section usually contains initialized data.
- **.sect** defines initialized named sections and associates subsequent code or data with that section. A section defined with .sect can contain code or data.
- **.text** identifies portions of code in the .text section. The .text section usually contains executable code.
- **.usect** 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.

The output listing on the next page shows how you can use sections directives to associate code and data with the proper sections. 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, its SPC resumes counting as if there had been no intervening code.

After the code in is assembled, the sections contain:

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

Note that 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.

```
 1                    ;*********************************
 2                    ;* Start assembling into .text    *
 3                    ;*********************************
 4 0000                     .text
 5 0000   01                .byte 1,2
   0001   02
 6 0002   03                .byte 3,4
   0003   04
 7
 8                    ;*********************************
 9                    ;* Start assembling into .data    *
10                    ;*********************************
11 0000                     .data
12 0000   09                .byte 9,10
   0001   0a
13 0002   0b                .byte 11,12
   0003   0c
14
15                    ;*********************************
16                    ;* Start assembling into named    *
17                    ;* section, var_defs              *
18                    ;*********************************
19 0000                     .sect "var_defs"
20 0000   11                .byte 17,18
   0001   12
21
22                    ;*********************************
23                    ;* Resume assembling into .data   *
24                    ;*********************************
25 0004                     .data
26 0004   0d                .byte 13,14
   0005   0e
27 0000                     .bss sym,19    ; reserve space in .bss
28 0006   0f                .byte 15,16    ; still in .data
   0007   10
29
30                    ;********************************
31                    ;* Resume assembling into .text  *
32                    ;********************************
33 0004                     .text
34 0004   05                .byte 5,6
   0005   06
35 0000               usym  .usect "xy",20 ; reserve space in xy
36 0006   07                .byte 7,8      ; still in .text
   0007   08
```

Example 4.1: Sections Directives

4.3 Directives That Initialize Memory

Several directives initialize memory:

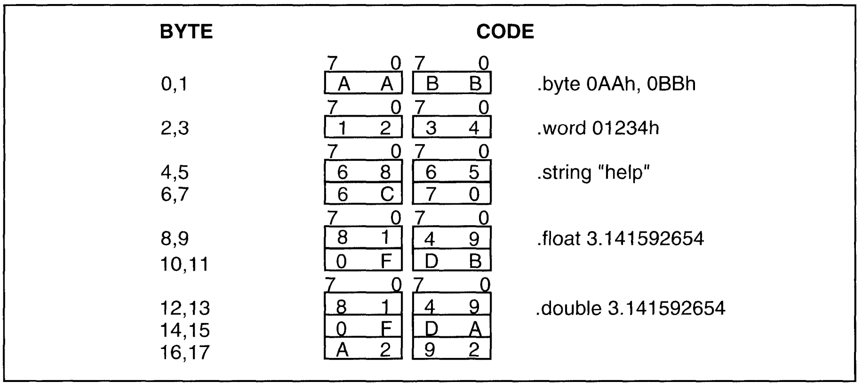
- **.byte** places one or more 8–bit values into consecutive bytes of the current section.
- **.word** places one or more 16–bit values into consecutive bytes in the current section.
- **.string** places 8–bit characters from one or more character strings into the current section. This directive is identical to .byte.
- **.float** calculates a (32–bit) MSP430 floating–point representation of a single precision floating–point value and stores it in four consecutive bytes in the current section.
- **.double** calculates a (48 bit) MSP430 floating-point representation of a double precision floating-point value and stores it in six consecutive bytes in the current section.

  The following code has been assembled for the example, that compares the .byte, .float, .word, and .string directives:

```
1 0000   aa                  .byte 0AAh, 0BBh
  0001   bb
2 0002   1234                .word 01234h
3 0004   68                  .string "help"
  0005   65
  0006   6c
  0007   70
4 0008   81490fdb            .float  3.141592654
5 000C   81490fdaa292        .double 3.141592654
```

Note: How the .byte, .word, .string, .float, .double and .field Directives Function in a .struct/.endstruct Sequence

The .byte, .word, .string, .float, .double and .field directives do not initalize memory when they are part of a .struct/.endstruct sequence; rather, they define a member´s size.

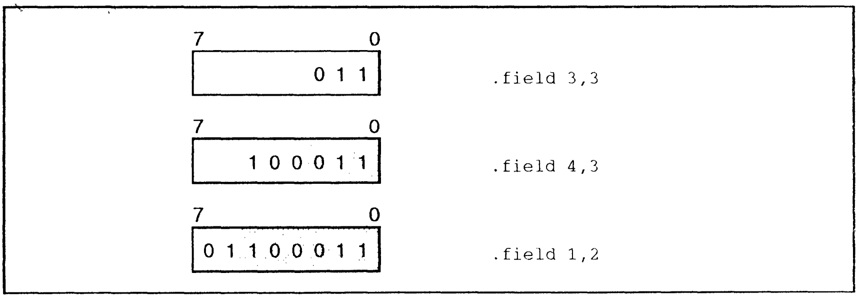


Example 4.2: Initialization Directives

- The **.field** directive places a single value into a specified number of bits in the current byte. With a field, you can pack multiple fields into a single byte; the assembler does not increment the SPC until a byte is filled.

  The next example shows how fields are packed into a byte. For this example, assume the following code has been assembled; note that the SPC doesn't change. (The fields are packed into the same byte.)

```
1      0000  03        .field   3,3
2      0000  23        .field   4,3
3      0000  63        .field   1,2
```



Example 4.3: The .field Directive

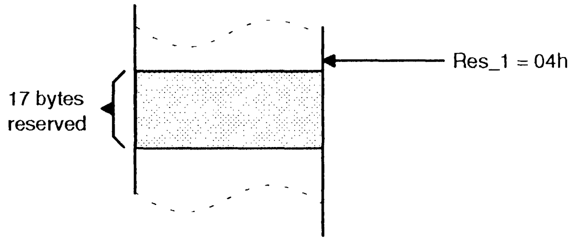
- The **.space** directive reserves a specified number of bytes in the current section. The assembler fills these reserved bytes with 0s.

  When you use a label with .space, it points to the *first* byte of the reserved block.

The following code has been assembled for the example of the .space directive:

```
45 0000   0100                   .word 100h,200h
   0002   0200
46 0004                Res_1:    .space 17
47 0016   000f                   .word 15
```

Res_1 points to the first byte in the space reserved by .space.

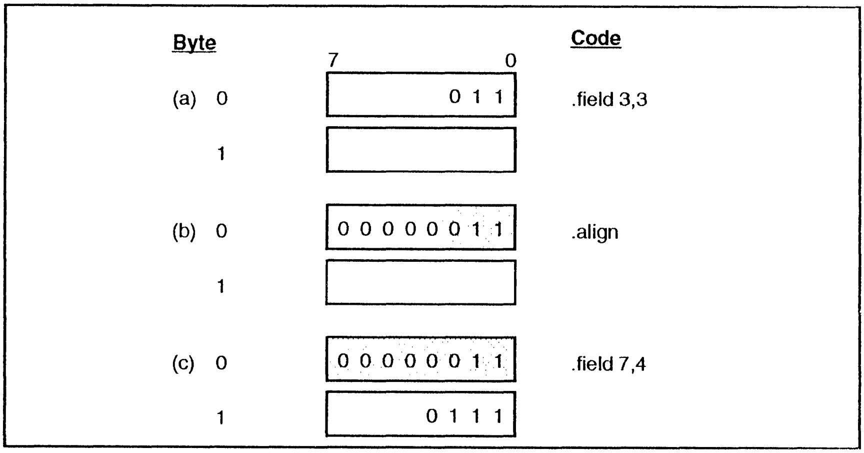


Example 4.4: The .space Directive

4.4 Directives That Align the Section Program Counter

- The .align directive aligns the SPC at the next byte boundary. This directive is useful with the .field directive when you do not wish to pack two adjacent fields in the same byte. The following code has been assembled for the example:

```
1 0000   03                    .field   3,3
2                              .align
3 0001   07                    .field   7,4
4
```



Example 4.5: The .align Directive

- The .even directive aligns the SPC at the next word boundary. It can be used to align bytes or strings on word boundaries:

```
1 0000   41                    .string "ABC"
  0001   42
  0002   43
2                              .even
3 0004   58                    .string "XYZ"
  0005   59
  0006   5a
4
```

4.5 Directives That Format the Output Listing

The following directives format the listing file:

- The source code contains a listing of 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 **.width** directive controls the page width 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 back on.
- The **.mlist** and **.mnolist** directives allow and inhibit macro expansion and loop block listings. You can use the .mlist directive to print all macro expansions and loop blocks to the listing.
- The **.option** directive controls several features in the listing file. This directive has several operands:

A	Turns on all listing (overrides all other directives and options).
B	Limits the listing of .byte directives to one line.
F	Resets the B, W, M, and T directives.
M	Turns off macro expansions in the listing.
T	Limits the listing of .string directives to one line.
W	Limits the listing of .word 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 -x option.)

- The **.page** directive causes a page eject in the output listing.
- The **.sslist** and **.ssnolist** directives allow and inhibit substitution symbol expansion listing. These directives are useful for debugging substitution symbols outside of macros.
- The **.title** directive supplies a title that the assembler prints at the top of each page.

4.6 Directives That Reference Other Files

These directives supply information for or about other files:

- 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 **.global** directive declares a symbol to be external so that it is available to other modules at link time. This directive does double duty, acting as a .def for defined symbols and as a .ref for undefined symbols. Note that the linker will resolve an undefined global symbol only if it is used in the program. The .global directive declares a symbol as 16 bits.
- The **.def** directive identifies a symbol that is defined in the current module and can be used by other modules. The assembler puts the symbol in the symbol table.
- The **.ref** directive identifies a symbol that is used in the current module but defined in another module. The assembler marks the symbol as an undefined external symbol and puts it in the object symbol table so that the linker can resolve its definition.
- 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.

4.7 Conditional Assembly Directives

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 *expression*	Marks the beginning of a conditional block and assembles code if the .if condition is true.
.elseif *expression*	Marks a block of code to be assembled if .if is false and .elseif is true.
.else	Marks a block of code to be assembled if .if is 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 *expression*	Marks the beginning a repeatable block of code.
.break *expression*	Continue to repeatedly assemble when the .break expression is false. Go to code immediately after .endloop if expression is true.
.endloop	Marks the end of a repeatable block.

The assembler supports several relational operators that are especially useful for conditional expressions.

4.8 Assembly–Time Symbol Directives

These directives equate meaningful symbol names to constant values or strings.

- 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
         .word   bval, bval*2, bval+12
         br      bval
  ```

  Note that the .set and .equ directives produce no object code.

- The **.struct/.endstruct** directives set up C–like structure definitions, and the **.tag** directive assigns the C–like structure characteristics to a label.

  The .struct./endstruct directives enable you to set up a C–like structure definition so that similar elements can be grouped together. Element offset calculation is then left up to the assembler. The .struct/.endstruct directives do not allocate memory. They simply create a symbolic template that can be used repeatedly.

  The .tag directive assigns structure characteristics to a label. 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     .byte
  y     .byte
  t_len .endstruct

  coord .tag   type          ; declare coord (coordinate)
        .usect coord,t_len   ;actual?memory?allocation
        add    coord.y,R4
  ```

- 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

        .byte coefficients
  ```

- The **.eval** directive evaluates an 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; for example:

  ```
        .asg   1 , x
        .loop
        .byte  x*10h
        .break   x = 4
        .eval  x+1, x
        .endloop
  ```

4.9 Miscellaneous Directives

This section discusses miscellaneous directives.

- The **.setsect** directive is generated by the absolute lister. It specifies an absolute starting address for a section name so that the assembler can generate an absolute listing.
- The **.setsym** directive is generated by the absolute lister. It specifies an absolute address for a global symbol. This allows the assembler to generate an absolute listing.
- The **.end** directive terminates assembly. It should be the last source statement of a program. This directive has the same effect as an end–of–file.
- The **.label** directive defines a special symbol that refers to the loadtime address rather than the runtime address within the current section.

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 way the assembler does, incrementing the error count and preventing the assembler from producing an object file.
- The **.wmsg** directive sends warning messages to the standard output device. The .wmsg directive functions in the same way the .emsg directive does, but increments the warning count and does not prevent 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 way the .emsg and .wmsg directives do, but does not set the error count or the warning count and does not prevent the assembler from producing an object file.

4.10 Directives Reference

The remainder of this chapter is a reference. Generally, the directives are organized alphabetically, one directive per page; related directives (such as .if/.else/.endif), however, are presented together on one page. Here's an alphabetical table of contents for the directives reference:

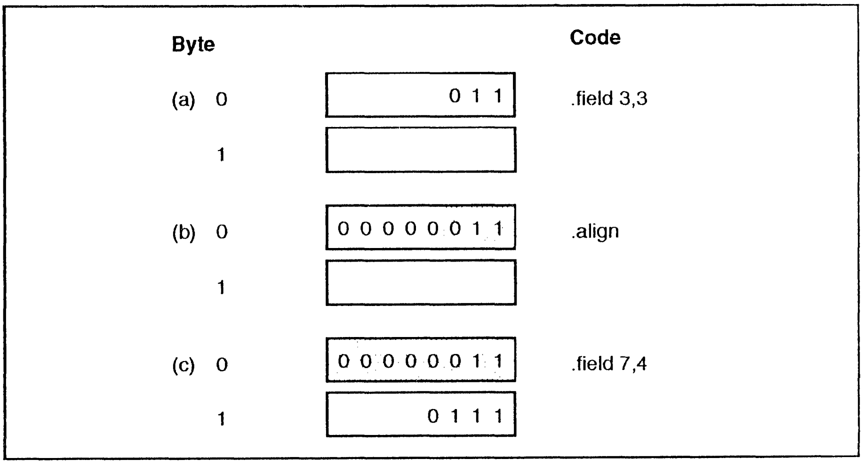
Directive	Page
.align	4-20
.asg	4-21
.break	4-46
.bss	4-23
.byte	4-24
.copy	4-25
.data	4-27
.double	4-28
.def	4-37
.elseif	4-40
.else	4-40
.emsg	4-29
.end	4-30
.endif	4-40
.endloop	4-46
.endstruct	4-61
.equ	4-56
.eval	4-21
.even	4-31
.fclist	4-32
.fcnolist	4-32
.field	4-33
.float	4-36
.global	4-37
.if	4-40
.include	4-25
.label	4-42
.length	4-43
.list	4-44
.loop	4-46
.mlib	4-48
.mlist	4-50
.mmsg	4-29
.mnolist	4-50
.newblock	4-51
.nolist	4-44
.option	4-52
.page	4-54
.ref	4-37
.sect	4-55
.set	4-56
.space	4-58
.sslist	4-59
.ssnolist	4-59
.string	4-24
.struct	4-61
.tag	4-61
.text	4-63
.title	4-64
.usect	4-65
.width	4-43
.wmsg	4-29
.word	4-67

Syntax **.align**

Description The .align directive aligns the section program counter (SPC) on the next byte boundary. This directive is useful with the .field directive when you do not wish to pack two adjacent fields in the same byte.

Example This example shows the creation of two fields that would normally be packed within the same byte. The .align directive forces these fields into separate bytes.

```
1 0000   03                          .field   3,3
2                                    .align
3 0001   07                          .field   7,4
4
```



Example 4.6: The .align Directive

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 (cannot be redefined) to a symbol, .asg assigns a character string (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* is a required parameter that must be a valid symbol name. The substitution symbol may be 32 characters long and must begin with a letter. Remaining characters of the symbol can be a combination of alphanumeric characters, underscores, and dollar signs.

The **.eval** directive performs arithmetic operations on substitution symbols. This directive evaluates the 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.

Example

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

```
         1                  .sslist; show expanded sub.syms
         2              ;*
         3              ;*  .asg/.eval example
         4              ;*
         5                         .asg     &, AND
         6                         .asg     R11, FP
         7
         8 0000 503b000c            add     #32 AND 44, FP
#                                   add     #32 & 44, R11
         9
        10                         .asg     0, x
        11                         .loop    5
        12                         .eval    x+1, x
        13                         .word    x
        14                         .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
```

Syntax **.bss** *name* [*, size in bytes, address*]

Description The .bss directive reserves space in the .bss section for variables. Use this directive to allocate space into RAM.

- The *name* is a required parameter. It defines a symbol that points to the first location reserved by the directive.
- The *size* is an optional parameter. It is a well–defined, absolute expression that specifies the number of bytes that are allocated. The default size for this directive is 1 byte.
- The *address* is an optional parameter that specifies a 16–bit address. It can be used only the first time a .bss directive is specified. Normally, the SPC is set to 0 the first time a named section is assembled; you can use the address parameter to assign an initial value to the SPC. This parameter has no effect on the final address of the section.

Section directives for initialized sections (.text, .data, and .sect) end the current section and begin assembling into another section. Section directives for uninitialized sections (.bss, .reg, .regpair, and .usect), however, do not affect the current section. The assembler assembles the .bss, .reg, .regpair, or .usect directive and then resumes assembling code into the same section.

Example This example shows the .bss directive used to allocate space for two variables, array and dflag. The symbol array points to 100 bytes of uninitialized space (at .bss–SPC = 0). The symbol dflag points to 1 byte of uninitialized space (at .bss–SPC = 100). Note that symbols declared with the .bss directive can be referenced in the same manner as other symbols and can also be declared global.

```
                ;*************************************
                ;* Begin assembling into .text      *
                ;*************************************
0000                 .text
0000  4a0b           mov   R10, R11

                ;*************************************
                ;* Allocate 100 bytes in .bss       *
                ;*************************************
0000                 .bss  array,100
0002  4b0c           mov   R11, R12 ; assembled into .text

                ;*************************************
                ;* Allocate 1 byte in .bss          *
                ;*************************************
0064                 .bss  dflag
0004 -4014005e       mov   dflag,R4 ;assembled into .text

                ;*************************************
                ;* Declare external .bss symbol     *
                ;*************************************
```

```
     22                 .global array  ; still in .text
```

Syntax

.byte *value*$_1$ [, ... , *value*$_n$]
.string *value*$_1$ [, ... , *value*$_n$]

Description

The .byte and .string directives place one or more 8–bit values into consecutive bytes of the current section. A *value* can be either :

- An expression that the assembler evaluates and treats as an 8–bit signed number, or
- A character string enclosed in double quotes. Each character in a string represents a separate value.

The assembler truncates values that are greater than 8 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 at which the assembler places the first byte.

Note that when you use .byte or .string in a .struct/.endstruct sequence, .byte or .string defines a member's size; it does not initialize memory.

Example

This example shows several 8–bit values placed into consecutive bytes in memory. 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 .string directive.

```
1 0000   0a          strx    .byte   10,-1,"abc",'a'
  0001   ff
  0002   61
  0003   62
  0004   63
  0005   61
2 0006   0a          stry    .string 10,-1,"abc",'a'
  0007   ff
  0008   61
  0009   62
  000a   63
  000b   61
```

Syntax

.copy ["]*filename*["]
.include ["]*filename*["]

(The quote marks surrounding the filename are optional.)

Description

The .copy and .include directives tell the assembler to read source statements from a different file. 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; the *filename* may be enclosed in double quotes. The *filename* must follow operating system conventions. You can specify a full pathname (for example, c:\430\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 -i assembler option.
3) Any directories specified by the environment variable A_DIR.

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 that are assembled.

The .copy and .include directives can be nested within a file being copied or included. The assembler limits this type of nesting to eight 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.

Example 1 This example shows how the .copy directive is used to tell the assembler to read and assemble source statements from other files, then to resume assembling into the current file.

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

Listing file:

```
         1 0000                          .space  29
         2                               .copy   "byte.asm"
A        1                         ;** In byte.asm
A        2 001d    20                    .byte 32,1+'A'
           001e    42
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    6a                    .byte 67h+3q
         3
         4                         ;** Back in original file
         5
         6 0025    64                    .string "done"
           0026    6f
           0027    6e
           0028    65
```

Example 2 This example shows how the .include directive is used to tell the assembler to read and assemble source statements from other files, then to resume assembling into the current file.

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

Listing file:

```
1 0000                  .space 29
2                       .include "byte2.asm"
3
4                       ;**Back in original file
5 0025    64            .string "done"
  0026    6f
  0027    6e
  0028    65
```

Syntax **.data** [*address*]

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.

The *address* is an optional parameter that specifies a 16–bit address. It can be used only the first time a .data directive is specified. Normally, the section program counter is set to 0 the first time the .data section is assembled; you can use this parameter to assign an initial value to the .data section program counter. This parameter has no effect on the final address of the section; it simply makes the listing easier to read.

Note that the assembler assumes that .text is the default section. Therefore, at the beginning of an assembly, the assembler assembles code into the .text section unless you specify an explicit section control directive.

Example This example shows the assembly of code into the .data and .text sections.

```
                ;**********************************
                ;**   Reserve space in .data     **
                ;**********************************
0000                     .data
0000                     .space 0cch

                ;**********************************
                ;**   Assemble into .text        **
                ;**********************************
0000                     .text
       00       Index    .equ   0
0000   4304               mov    #Index, R4

                ;**********************************
                ;**   Assemble into .data        **
                ;**********************************
00cc            Table:   .data
00cc   ffff              .word  -1
00ce   ff                .byte  0ffh

                ;**********************************
                ;**   Assemble into .text        **
                ;**********************************
0002                     .text
0002  "901400c8          cmp Table, R4

                ;**********************************
                ;**   Resume Assembling into .data **
                ;**********************************
00cf                     .data
```

Syntax **.double** value

Description The .double directive places the floating-point representation of a double floating-point constant into six bytes in the current section. The value must be a floating-point constant. Each constant is converted to a floating-point value in the MSP430 (48-bit) format.

The 48-bit value consists of three fields:

- An 8-bit biased exponent (e)
- A 1-bit sign field (s)
- A 39-bit fraction (f)

The value is stored exponent byte first, most significant byte of fraction second, and least significant byte of fraction sixth in the following format:

47 40	**39**	**38 0**
e	s	f

Note that when you use .double in a .struct/.endstruct sequence, .double defines a member's size; it does not initialize memory. For more information about MSP430 floating-point format, refer to Appendix G.

Examples Here are some examples of the .double directive.

```
0000    d38459516140          .double  -1.0e25
0006    814000000000          .double  3
000c    8d40e4000000          .double  12345
```

Syntax

.emsg *string*

.mmsg *string*

.wmsg *string*

Description

Use these directives to define your own error and warning messages. Note that 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 error messages to the standard output device. The .emsg directive generates errors in the same way the assembler does, incrementing the error count and preventing the assembler from producing an object file.
- The **.wmsg** directive sends warning messages to the standard output device. The .wmsg directive functions in the same way the .emsg directive does, but increments the warning count. The assembler is not prevented from producing an object file.
- The **.mmsg** directive sends assembly–time messages to the standard output device. The .mmsg directive functions in the same way the .emsg and .wmsg directives do, but does not set the error count or the warning count. The assembler is not prevented from producing an object file.

Example

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

```
  1         MSG_EX  .macro parm1
  2                 .if    $symlen(parm1) = 0
  3                 .emsg  "ERROR -- MISSING PARAMETER"
  4                 .else
  5                 MOV    parm1, A
  6                 .endif
  7                 .endm
  8
  9 0000            MSG_EX
1                   .if    $symlen(parm1) = 0
1                   .emsg  "ERROR -- MISSING PARAMETER"
1                   .else
1                   MOV    parm1, A
1                   .endif
```

These messages will show in the readout like any other error message:

```
********* USER ERROR - ERROR -- MISSING PARAMETER
********* USER WARNING - . . .
********* USER MESSAGE - . . .
```

Syntax **.end**

Description The .end directive is an optional directive that terminates assembly. It should be the last source statement of a program. The assembler ignores any source statements that follow an .end directive.

This directive has the same effect as an end–of–file. You can also use .end when you're debugging code and you'd like to stop assembling at a specific point in your code.

> **Note: Use .endm to End a Macro**
> 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.

```
1 0000                   Text_start:     .text
2 0000   0a                              .byte 0ah
3 0002   aaaa                            .word 0aaaah
4 0004   61                              .string "aaa"
  0005   61
  0006   61
5                                        .end
```

Syntax **.even**

Description The .even directive aligns the section program counter (SPC) on the next word boundary. This directive can be used to force the start of the next initialized data on an even address.

Example This example shows the initialization of two strings, each aligned on a word boundary. Without the .even directive the second string would start immediately after the first one (on an odd address).

```
 1 0000  41          .string     "ABC"
0001  42
0002  43
 2                   .even
 3 0004  58          .string     "XYZ"
   0005  59
0006  5a
 4
```

Note: Automatic Alignment on Word Boundary

Instructions itself and data created by .word, .float and .double directives will be aligned on a word boundary automatically.

Syntax **.fclist**

.fcnolist

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

- The **.fclist** directive allows the listing of conditional blocks that do not produce code (false blocks). *By default, the assembler behaves as if you had used .fclist.*
- The **.fcnolist** directive inhibits the listing of false conditional blocks that do not produce code. Only code in the conditional block that actually assembles appears in the listing. The .if, .elseif, .else, and .endif directives do not appear.

Example This example shows the assembly language file and the listing file for code with and without the conditional blocks listed. This is the unassembled file:

```
x   .set 1          ;True
y   .set 0          ;False

    .fclist

    .if x
    MOV #5, R4
    .else
    MOV #9, R4
    .endif

    .fcnolist

    .if x
    MOV #5, R4
    .else
    MOV #9, R4
    .endif
```

This is the listing file:

```
 1         01             x    .set 1    ;True
 2         00             y    .set 0    ;False
 3
 4                             .fclist
 5
 6                             .if x
 7 0000    40340005            MOV #5, R4
 8                             .else
 9                             MOV #9, R4
10                             .endif
11
12                             .fcnolist
13
15 0004    40340005            MOV #5, R4
```

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

Description The .field directive initializes multiple–bit fields within a single word 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* is an optional parameter; it specifies a number from 1 to 16, which is the number of bits the field consists of. If you do not specify a size, the assembler assumes that the size is 16 bits. If you specify a value that cannot fit in *size* 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 - value truncated.
```

Successive .field directives pack values into the specified number of bits in the current word. Fields are packed starting at the least significant part of the word, moving toward the most significant part as more fields are added. If the assembler encounters a field size that does not fit into the current word, it writes out the word, increments the SPC, and begins packing fields into the next word.

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

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

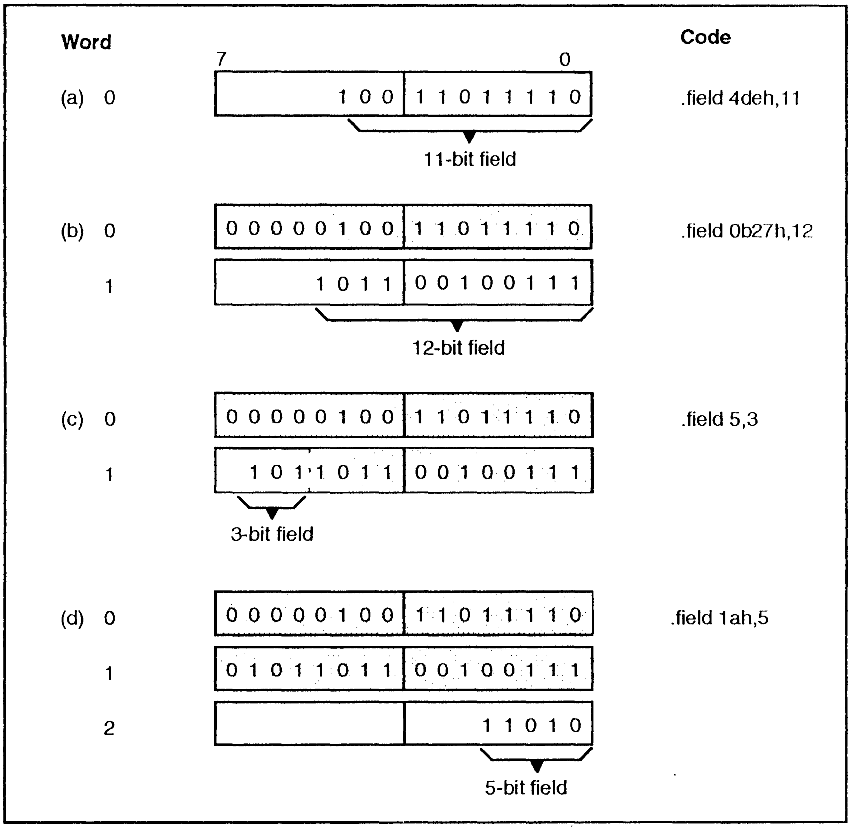
Note also that when you use .field in a .struct/.endstruct sequence, .field defines a member's size; it does not initialize memory.

Example

This example shows how fields are packed into a word. Note that the SPC does not change until a word is filled and the next word is begun.

```
                         ********************************
                         *   Initialize a 11-bit field  *
                         ********************************
 0000   04de                     .field  4deh,11

                         ********************************
                         *   Initialize a 12-bit field  *
                         *   in a new word              *
                         ********************************
 0000   0b27                     .field  0b27h,12

                         ********************************
                         *   Initialize a 3-bit field   *
                         *   in the same byte           *
                         ********************************
 0000   5b27                     .field  5,3

                         ********************************
                         *   Initialize a 5-bit field   *
                         *   in the next byte           *
                         ********************************
 0000   1a                       .field 1ah,5
```

The example shows how the directives affect memory.



Example 4.7: The .field Directive

Syntax **.float** *value*

Description

The .float directive places the floating–point representation of a single floating–point constant into four bytes in the current section. The *value* must be a floating–point constant. Each constant is converted to a floating–point value in MSP430 (32–bit) format.

The 32–bit value consists of three fields:

- An 8–bit biased exponent (*e*)
- A 1–bit sign field (*s*)
- A 23–bit fraction (*f*)

The value is stored exponent byte first, most significant byte of fraction second, and least significant byte of fraction fourth in the following format:

31 **24**	**23**	**22** **0**
e	s	f

Note that when you use .float in a .struct/.endstruct sequence, .float defines a member's size; it does not initialize memory. For more information about MSP430 floating–point format, refer to Appendix G.

Example

Here are some examples of the .float directive.

```
1 0000   d3845951                .float   -1.0e25
2 0004   81400000                .float   3
3 0008   8d40e400                .float   12345
```

Syntax

.global *symbol$_1$* [, ... , *symbol$_n$*]

.def *symbol$_1$* [, ... , *symbol$_n$*]

.ref *symbol$_1$* [, ... , *symbol$_n$*]

Description

The .global, .def, and .ref directives identify global symbols, which 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 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. Note that .ref always creates an entry for a symbol, whether the module uses the symbol or not; .global, however, create a symbol table entry only if the module actually uses the symbol.

A symbol may be declared global for two reasons:

1) If the symbol is *not defined in the current module* (including 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.

2) If the symbol *is defined in the current module*, the .global, .globreg, 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:

- file1.lst and file3.lst 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. file1.lst uses the .global directive to identify the global symbols; file3.lst uses .ref and .def to identify the symbols.
- file2.lst and file4.lst 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. file2.lst uses the .global directive to

identify the nonregister global symbols; file4.lst uses .ref and .def to identify the nonregister symbols.

file1.lst

```
 1                     ; Global symbol defined in this file
 2                              .global init
 3
 4                     ; Global symbols defined in file2.lst
 5                              .global x, y, z
 6
 7 0000   5034002c  init:    add #44, R4
 8 0004  !0000                  .word x
 9                     ;  .
10                     ;  .
11                     ;  .
12                               .end
```

file2.lst

```
 1                     ; Global symbol defined in this file
 2                               .global x, y, z
 3
 4                     ; Global symbols defined in file1.lst
 5                               .global init
 6
 7         01          x       .equ    1
 8         02          y       .equ    2
 9         03          z       .equ    3
10 0000  !0000                  .word   init
11                     ;  .
12                     ;  .
13                     ;  .
14                             .end
```

file3.lst

```
 1                   ; Global symbol defined in this file
 2                            .def init
 3
 4                   ; Global symbols defined in file4.lst
 5                            .ref x, y, z
 6
 7 0000   5034002c  init:   add #44, R4
 8 0004  !0000                .word x
 9                   ;  .
10                   ;  .
11                   ;  .
12                            .end
```

file4.lst

```
 1                   ; Global symbol defined in this file
 2                            .def x, y, z
 3
 4                   ; Global symbols defined in file3.lst
 5                            .ref init
 6
 7        01         x       .equu    1
 8        02         y       .equ    2
 9        03         z       .equ    3
10 0000  !0000                .word   init
11                   ;  .
12                   ;  .
13                   ;  .
14                            .end
```

Syntax

.if *well–defined expression*
code to assemble when the expression is true

.elseif *well–defined expression*
code block to execute when the expression is true

.else
code to assemble when the expression is false

.endif
terminate condition block

Description

Four directives provide conditional assembly:

- The **.if** directive marks the beginning of a conditional block. The *expression* is a required parameter.
  - If the expression evaluates to *true* (nonzero), the assembler assembles the code that follows it (up to an .elseif, an .else, or an .endif).
  - If the expression evaluates to *false* (0), the assembler assembles code that follows an .elseif (if present), an else (if present), or an .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 an .endif. 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 an .else (if present) or an .endif.
- The **.else** directive identifies a block of code that the assembler assembles when the .if expression is false (0). This 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.

Example Here are some examples of conditional assembly:

```
 1        01    sym1  .set     1
 2        02    sym2  .set     2
 3        03    sym3  .set     3
 4        04    sym4  .set     4
 5              If_4: .if   sym4=sym2*sym2
 6 0000   04          .byte sym4        ;  Equal values
 7                    .else
 8                    .byte sym2 * sym2 ; Unequal values
 9                    .endif
10              If_5: .if   sym1<=10
11 0001   0A          .byte 10          ; Less than/equal
12                    .else
13                    .byte sym1        ; Greater than
14                    .endif
15              If_6: .if   sym3*sym2!=sym4+sym2
16                    .byte sym3*sym2   ;Unequal values
17                    .else
18 0002   06          .byte sym4+sym2   ;Equal values
19                    .endif
20              If_7  .if   sym1=2
21                    .byte sym1
22                    .elseif sym2+sym3=5
23 0003   05          .byte sym2+sym3
24                    .endif
```

Syntax **.label** *symbol*

Description

The .label directive defines a special symbol that refers to the loadtime address rather than the runtime address within the current section. Most sections created by the assembler have relocatable addresses. The assembler assembles each section as if it started at zero, and the linker relocates it to the address at which it loaded and ran.

For some applications, it is desirable to have a section load at one address and run at a **different** address. For example, you may wish to load a block of performance–critical code into slower off–chip memory to save space, and then move the code to high–speed on–chip memory to run it.

Such a section is assigned two addresses at link time: a load address and a separate run address. All labels defined in the section are relocated to refer to the runtime 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 *loadtime* address. This is useful primarily so that the code that relocates the section knows where the section was loaded. For example:

```
;---------------------------------------------
; .label Example
;---------------------------------------------
      .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
```

Syntax

.length *page length*
.width *page width*

Description

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

- Default length: 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 following lines; you can reset the page width with another .width directive.

- Default width: 132 characters
- Minimum width: 80 characters
- Maximum width: 200 characters

Note that 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
```

Syntax

.list
.nolist

Description

The .nolist directive suppresses the source listing output until a .list directive is encountered. The .list directive tells the assembler to resume printing the source listing after it has been stopped by a .nolist directive. *By default, the assembler acts as if a .list directive had been specified.* The .nolist directive can be used to reduce assembly time and the size of the source listing; it can be used in macro definitions to inhibit 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. At the beginning of an assembly, the assembler acts as if it has assembled a .list directive.

Note: Creating a Listing File (-l option)

If you don't request a listing file when you invoke the assembler, the assembler ignores the .list directive.

Example

This example shows how to use the .copy directive to insert 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. Note that the .nolist, the second .copy, and .list directives do not appear in the listing file; note also that the line counter is incremented even when source statements are not listed.

Source file:

```
       .copy    "copy2.asm"
* Back in original file
       .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
        6                     * Back in original file
        7 0008    44                 .string "Done"
          0009    6f
          000a    6e
          000b    65
```

Syntax

.loop [*well–defined expression*]
code block to repeatedly assemble

.break [*well–defined expression*]
continue to assemble repeatedly when the .break expression is false (zero); go to code immediately following .endloop if expression is true (nonzero)

.endloop
code block to execute when the .break directive is true (nonzero) or when the .break expression is omitted and the loop count equals the expression

Description

Three directives enable 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. If there is no expression, the loop count defaults to 1024, unless the assembler encounters a .break directive.
- The **.break** directive is optional, along with its expression. When the expression is false (0), the loop continues. When the expression is true (nonzero) or omitted, the assembler breaks the loop and assembles the code after the .endloop directive.
- The **.endloop** directive terminates a repeatable block of code.

Example This example illustrates how these directives can be used with the .eval directive.

```
  1                               .eval   0, x
  2                        coef   .loop
  3                               .word   x*100
  4                               .eval   x+1,x
  5                               .break  x = 7
  6                               .endloop
1         0000   0000             .word   0*100
1                                 .eval   0+1,x
1                                 .break  1 = 7
1         0002   0064             .word   1*100
1                                 .eval   1+1,x
1                                 .break  2 = 7
1         0004   00c8             .word   2*100
1                                 .eval   2+1,x
1                                 .break  3 = 7
1         0006   012c             .word   3*100
1                                 .eval   3+1,x
1                                 .break  4 = 7
1         0008   0190             .word   4*100
1                                 .eval   4+1,x
1                                 .break  5 = 7
1         000a   01f4             .word   5*100
1                                 .eval   5+1,x
1                                 .break  6 = 7
1         000c   0258             .word   6*100
1                                 .eval   6+1,x
1                                 .break  7 = 7
```

Syntax **.mlib** ["]*filename*["]

(The quote marks surrounding the filename are optional.)

Description The .mlib directive provides the assembler with the name of a macro library. A macro library is a collection of files that contain macro definitions. These files are bound into a single file (called a library or archive) by the archiver. Each member of a macro library may contain one macro definition that corresponds to the name of the file. Note that:

- Macro library members must be **source** files (not object files).
- 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 may be enclosed in double quotes. You can specify a full pathname (for example, c:\430\macs.lib). 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 -i assembler option.
3) Any directories specified by the environment variable A_DIR .

When the assembler encounters an .mlib directive, it opens the library 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 manner as 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.

Example This example shows how to create a macro library that defines two macros, inc1 and dec1. The file inc1.asm contains the definition of inc1, and dec1.asm contains the definition of dec1.

inc1.asm	**dec1.asm**
`* Macro for incrementing`	`* Macro for decrementing`
`inc1   .MACRO nam`	`dec1   .MACRO nam`
`       mov     nam, R4`	`       mov     nam, R4`
`       inc     R4`	`       dec     R4`
`       mov     R4, nam`	`       mov     R4, nam`
`       .ENDM`	`       .ENDM`

```
ar430 -a mac inc1.asm dec1.asm
```

Now you can use the .mlib directive to reference the macro library and define the inc1 and dec1 macros:

```
        1 0000                .bss     var1,1
        2 0001                .bss     var2,1
        3
        4                     .mlib    "mac.lib"
        5
        6 0000                inc1     var1    ; macro call
1         0000  -40140000     mov     var1, R4
1         0004   5314         inc     R4
1         0006  -44800000     mov     R4, var1
        7 000a                dec1     var2    ; macro call
1         000a  -4014fff5     mov     var2, R4
1         000e   8314         dec     R4
1         0010  -4480ffef     mov     R4, var2
```

Syntax

.mlist
.mnolist

Description

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

- The **.mlist** directive allows macro and .loop/.endloop block expansions in the listing file.
- The **.mnolist** directive inhibits macro and .loop/.endloop block expansions in the listing file.

By default, all code encountered in macros and .loop/.endloop blocks is listed.

Example

This example shows how to define 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 .mnolist directive was assembled. The third time the macro is called, the macro expansion is again listed because a .mlist directive was assembled.

```
 1             str_3   .MACRO  pm1, pm2, pm3
 2                     .string ":pm1:", ":pm2:", ":pm3:"
 3                     .ENDM
 4
 5 0000                str_3   "as","I","am"
1  0000   61           .string "as", "I", "am"
   0001   73
   0002   49
   0003   61
   0004   6d
 6                     .mnolist
 7 0005                str_3   "as","I","am"
 8                     .mlist
 9 000a                str_3   "as","I","am"
1  000a   61           .string "as", "I", "am"
   000b   73
   000c   49
   000d   61
   000e   6d
```

Syntax **.newblock**

Description The .newblock directive undefines any local labels currently defined. A local label, by nature, is temporary; the .newblock directive resets local labels and terminates their scope.

A local label is a label in the form $*n*, where *n* is a single decimal digit. A local label, like other labels, points to an instruction byte. Unlike other labels, local labels cannot be used in expressions; they can be used only as the operand in 10-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. Note that the .text, .data, and .sect directives also reset local labels and that local labels that are defined within an include file are not valid outside of the include file.

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

```
0000   4c0d      Label1: mov     R12, R13
0002   2000              jnz     $1
0004   433d              mov     #-1, r13
0006   9d04      $1      cmp     R13, R4
                         .newblock       ;undefine $1
0008   2000              jne     $1
000a   531d              inc     R13
000c   5d0e      $1      add     R13, R14
```

Syntax	**.option** *option list*
Description	The .option directive selects several options for the assembler output listing. The *option list* is a list of options separated by commas; each option selects a listing feature. Valid options include:

A	Turns on all listing (overrides all other directives and options).
B	Limits the listing of .byte directives to one line.
F	Resets the B, M, W, and T options.
M	Turns off macro expansions in the listing.
T	Limits the listing of .string directives to one line.
W	Limits the listing of .word directives to one line.
X	Produces a symbol cross–reference listing.

Options **are not** case–sensitive.

Example This example shows how to limit the listings of the .byte, .word, and .string directives to one line each.

```
 1                 ;***************************************
 2                 ;*  Limit the listing of .byte, .word,  *
 3                 ;*  .string directives to 1 line each   *
 4                 ;***************************************
 5                          .option B, W, T
 6 0000    bd               .byte   -'C', 0B0h, 5
 7 0004    15aa             .word   5546, 78h
 8 0008    59               .string "YES"
 9                 ;***************************************
10                 ;* Reset the listing options            *
11                 ;***************************************
12                          .option F
13 000b    bd               .byte   -'C', 0B0h, 5
   000c    b0
   000d    05
14 000e    15aa             .word   5546, 78h
   0010    0078
15 0012    59               .string "YES"
   0013    45
   0014    53
16                 ;***************************************
17                 ;*  Use The A option to ignore all      *
18                 ;*  other options and directives        *
19                 ;***************************************
20                          .option A
21                          .nolist
22                          .option B, W, T
23 0015    bd               .byte   -'C', 0B0h, 5
   0016    b0
   0017    05
24 0018    15aa             .word   5546, 78h
   001a    0078
25 001c    59               .string "YES"
   001d    45
   001e    53
26                          .list
```

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 line counter is incremented. 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:

```
MSP TSS430 Macro Assembler Prototype Version 1.0 [Mar 22] Wed Aug 18 08:34:18 1993
 Copyright (c) 1993    Texas Instruments Incorporated

**** Page Directive Example ****                                       PAGE    1

        2                        ;         .
        3                        ;         .
        4                        ;         .
MSP TSS430 Macro Assembler Prototype Version 1.0 [Mar 22] Wed Aug 18 08:34:18 1993
 Copyright (c) 1993    Texas Instruments Incorporated

**** Page Directive Example ****                                       PAGE    2
```

Syntax **.sect** "*section name*"[,*address*]

Description The .sect directive defines a named section that can be used like the default .text and .data sections. The .sect directive begins assembling source code into the named section.

- The *section name* identifies a section that the assembler assembles code into. The *name* is significant to 8 characters and must be enclosed in double quotes.
- The *address* is an optional parameter that specifies a 16–bit address. It can be used only the first time a .sect directive is specified for a particular section. Normally, the SPC is set to 0 the first time a named section is assembled; you can use the address parameter to assign an initial value to the SPC. This parameter has no effect on the final address of the section; it simply makes the listing easier to read.

Example This example shows how two special–purpose sections, Sym_Defs and Vars, are defined, and how code assembles into them.

```
 1                        ;*************************************
 2                        ;*  Begin assembling into .text       *
 3                        ;*************************************
 4 0000                             .text
 5 0000   4b0c                      MOV R11, R12
 6 0002   4d0e                      MOV R13, R14
 7
 8                        ;*************************************
 9                        ;*  Begin assembling into Sym_Defs    *
10                        ;*************************************
11 0000                             .sect   "Sym_Defs"
12 0000   00aa      X               .word   0aah
13 0002   50340005                  ADD     #5, R4
14
15                        ;*************************************
16                        ;*  Begin assembling into Vars        *
17                        ;*************************************
18 4000                             .sect "Vars", 4000h
19        10        Word_Len        .set  16
20        08        Byte_Len        .set  16 / 2
21
22                        ;*************************************
23                        ;*  Resume assembling into .text      *
24                        ;*************************************
25 0004                             .text
26 0004   5034002a                  ADD #42, R4
27 0008   03                        .byte 3,4
          0009   04
28                        ;*************************************
29                        ;*  Resume assembling into Vars       *
30                        ;*************************************
31 4000                             .sect "Vars"
32 4000   01                        .field 1,2
33 4000   09                        .field 2,2
```

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* 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.

Example This example shows how symbols can be assigned with .set.

```
                   ;***********************************
                   ;* Equate symbol FP to register R11  *
                   ;*  and use it instead of R11        *
                   ;***********************************
        0b         FP      .set    R11
   0000 4b24               MOV     @FP, R4

                   ;***********************************
                   ;* Set symbol count to an integer    *
                   ;* expression and use it as an       *
                   ;* immediate operand                 *
                   ;***********************************
        35         count   .equ    100/2+3
   0002 40340035           MOV     #count, R4

                   ;***********************************
                   ;* Set symbol symtab to relocatable  *
                   ;* expression                        *
                   ;***********************************
   0006 000a       label   .word   10
       '07         symtab  .set    label+1
   0008'0007               .word   symtab

                   ;***********************************
                   ;* Set symbol nsyms to another       *
                   ;* symbol (count) and use it instead *
                   ;* of count                          *
                   ;***********************************
        35         nsyms   .equ    count
   000a 40340035            MOV     #nsyms, R4
```

Syntax

.space *size in bytes*

Description

The .space directive reserves ***size*** number of bytes in the current section and fills them with 0s. The section program counter is incremented to point to the byte following the reserved space.

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

Example

This example shows how the .space directive reserves memory.

```
 1                         ;**********************************
 2                         ;*  Begin assembling into .text    *
 3                         ;**********************************
 4 0000                              .text
 5
 6                         ;**********************************
 7                         ;*  Reserve 15 bytes in .text      *
 8                         ;**********************************
 9 0000                              .space 0fh
10 0010   0100                       .word 100h, 200h
   0012   0200
11
12                         ;**********************************
13                         ;*  Begin assembling into .data    *
14                         ;**********************************
15 0000                               .data
16 0000   2e                          .string ".data"
   0001   64
   0002   61
   0003   74
   0004   61
17
18                         ;**********************************
19                         ;*  Reserve 100 bytes in .data;    *
20                         ;*  Res_1 points to the first byte *
21                         ;**********************************
22 0005                    Res_1     .space 100
23 006a   000f                       .word  15
24 006c  "0005                       .word  Res_1
```

Syntax **.sslist**

.ssnolist

Description Two directives enable 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 inhibits substitution symbol expansion in the listing file.

By default, all substitution symbol expansion in the listing file is inhibited. The lines with the pound (#) character denote expanded substitution symbols.

Example

This example shows code that by default (.ssnolist directive) inhibits the listing of substitution symbol expansion, and it shows the .sslist directive assembled, which tells the assembler to list substitution symbol code expansion.

```
       1 0000                                  .bss     x,1
       2 0001                                  .bss     y,1
       3
       4                             ADD2      .macro   pm1, pm2
       5                                       mov      pm1,R4
       6                                       mov      R4, R5
       7                                       mov      pm2, R4
       8                                       add      R5, R4
       9                                       mov      R4, pm2
      10                                       .endm
      11
      12                                       .asg     R11, FP
      13                                       .asg     R13, SSP
      14
      15 0000   4b0d                           mov      FP, SSP
      16 0002                                  add2     x, y
1           0002  -40140000                    mov      x,R4
1           0006   4405                        mov      R4, R5
1           0008  -4014fff7                    mov      y, R4
1           000c   5504                        add      R5, R4
1           000e  -4480fff1                    mov      R4, y
      17
      18                                       .sslist
      19
      20 0012   4b0d                           mov      FP, SSP
#                                              mov      R11, R13
      21 0014                                  add2     x, y
1           0014  -40140000                    mov      pm1,R4
#                                              mov      x,R4
1           0018   4405                        mov      R4, R5
1           001a  -4014ffe5                    mov      pm2, R4
#                                              mov      y, R4
1           001e   5504                        add      R5, R4
1           0020  -4480ffdf                    mov      R4, pm2
#                                              mov      R4, y
```

Syntax

[*stag*$_1$]	**.struct**		[*expr*$_1$]
[*mem*$_0$]	*element*		[*expr*$_2$]
[*mem*$_1$]	*element*		[*expr*$_2$]
.	.		.
.	.		.
[*mem*$_n$]	**.tag**	*stag*$_2$	[,*expr*$_2$]
.	.		.
[*mem*$_N$]	*element*		[*expr*$_2$]
[*size*]	**.endstruct**		
label	**.tag**		*stag*$_1$

Description

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

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

[*stag*$_1$]	Is the structure's tag. Its value is associated with the beginning of the structure. If no stag is present, this tells the assembler to put the structure members in the global symbol table with their value being their absolute offset from the top of the structure.
[*expr*$_1$]	Is an expression indicating the beginning offset of the structure. Structures default to start at 0.
[*mem*$_n$]	Is a label for a member of the structure. This label is absolute and equates to the present offset from the beginning of the structure.
element	Is one of the following descriptors: .string, .byte, .word, .float, .double, .tag, and .field. 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. A .tag directive is a special case because a stag must be specified (such as stag$_2$ in the definition).
[*expr*$_2$]	Is an expression for the number of elements described. This value defaults to 1.
[*size*]	Is a label for the total size of the structure.

Note: The Types of Directives That Can Appear in a .struct/.endstruct Sequence

The only directives that can appear in a .struct/.endstruct sequence are element descriptors, conditional assembly directives, and the .align and .even directives, which align the member offsets on byte resp. word boundaries. Note that empty structures are illegal.

Examples

```
 1       0000    real_rec  .struct         ;stag
 2       0000    nom       .byte           ;member1  = 0
 3       0001    den       .byte           ;member2  = 1
 4       0002    real_len  .endstruct      ;real_len = 2
 5
 6 0000  -4014ffff         mov   real+real_rec.den,R4      ;access
 7
 8 0000                    .bss  real,real_len             ;allocate
 9

10       0000    cplx_rec  .struct         ;stag
11       0000    reali     .tag  real_rec  ;member1  = 0
12       0002    imagi     .tag  real_rec  ;member2  = 2
13       0004    cplx_len  .endstruct      ;cplx_len = 4
14
15               complex   .tag  cplx_rec
16 0002                    .bss  complex,cplx_len          ;allocate
17
18 0004  -4014fffe         mov   complex.imagi.nom,R4
19 0008  -9014fff9         cmp   complex.reali.den,R4
20

21       0000              .struct         ;no stag puts members
22                                        ;into global symbol table
23       0000    x         .byte           ;create 3dim templates
24       0001    y         .byte
25       0002    z         .byte
26                         .endstruct
27

28       0000    bit_rec   .struct         ;stag
29       0000    stream    .string 64
30       0040    bit7      .field  7
31       0040    bit2      .field  2
32       0041    bit5      .field  5
33       0042    x_int     .byte
34       0043    bit_len   .endstruct
35
36               bits      .tag  bit_rec
37 0006                    .bss  bits,bit_len
38
39 000c  -40140038         mov   bits.bit7,R4 ;load field
40 0010  f034007f          and   #7fh,R4      ;mask off garbage
```

Syntax **.text** [*address*]

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 *address* is an optional parameter that specifies a 16–bit address. It can be used only the first time a .text directive is specified. Normally, the section program counter is set to 0 the first time the .text section is assembled; you can use this parameter to assign an initial value to the .text section program counter. This parameter has no effect on the final address of the section; it simply makes the listing easier to read.

Note that the assembler assumes that .text is the default section. Therefore, at the beginning of an assembly, the assembler assembles code into the .text section unless you specify one of the other sections directives (.data or .sect).

Example This example shows code assembled into the .text and .data sections. The .data section contains integer constants, and the .text section contains character strings.

```
 1                  ;************************************
 2                  ;* Begin assembling into .data      *
 3                  ;************************************
 4 0000                           .data
 5 0000   05                      .byte 5,6
   0001   06
 6
 7                  ;************************************
 8                  ;* Begin assembling into .text      *
 9                  ;************************************
10 7000                           .text 7000h
11 7000   01                      .byte 1
12 7001   02                      .byte 2, 3
   7002   03
13
14                  ;************************************
15                  ;* Resume assembling into .data     *
16                  ;************************************
17 0002                           .data
18 0002   07                      .byte 7,8
   0003   08
19
20                  ;************************************
21                  ;* Resume assembling into .text     *
22                  ;************************************
23 7003                           .text
24 7003   04                      .byte 4
```

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 65 characters. If you supply more than 65 characters, the assembler truncates the string and issues a warning.

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 of a listing, the first source statement must contain a .title directive.

Example This example shows how to print one title on the first page and a different title on succeeding pages.

Source file:

```
        .title   "** Integer Routines **"
;         .
;         .
;         .
        .title   "** Floating Point Routines **"
        .page
```

Listing file:

```
MSP TSS430 Macro Assembler Prototype Version 1.0 [Mar 22] Wed Aug 18 11:01:02 1993
 Copyright (c) 1993    Texas Instruments Incorporated

** Integer Routines **                                              PAGE    1

        2                         ;         .
        3                         ;         .
        4                         ;         .
MSP TSS430 Macro Assembler Prototype Version 1.0 [Mar 22] Wed Aug 18 11:01:02 1993
 Copyright (c) 1993    Texas Instruments Incorporated

** Floating Point Routines **                                       PAGE    2
```

Syntax *symbol* **.usect** "*section name*", *size in bytes*

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 have no contents. .usect defines additional sections, however, that can be placed anywhere in memory, independently of the .bss section.

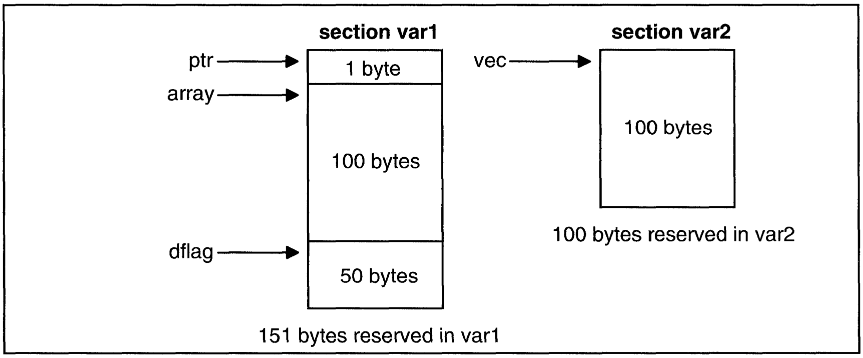
- The *symbol* points to the first location reserved by this invocation of the .usect directive. The *symbol* corresponds to the name of the variable that you're reserving space for.
- The *section name* must be enclosed in double quotes; only the first 8 characters are significant. This parameter names the uninitialized section.
- The *size* is an expression that defines the number of bytes that are reserved in section *name.*
- The *address* is an optional parameter that specifies a 16–bit address. It can be used only the first time a .usect directive is specified for a particular section. Normally, the SPC is set to 0 the first time a named section is assembled; you can use the address parameter to assign an initial value to the SPC. This parameter has no effect on the final address of the section.

Other sections directives (.text, .data, and .sect) end the current section and tell the assembler to begin assembling into another section. The .usect, .bss, .regpair, and .reg directives, however, do not affect the current section. The assembler assembles the .usect, .bss, .regpair, and .reg directives and then resumes assembling into the current section.

You can repeat the .usect directive to define more than one variable in the specified section. Variables that can be located contiguously in memory can be defined in the same section by using multiple .usect directives with the same section name.

Example This example shows how to use 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.

```
                        ;**********************************
                        ;*  Assemble into .text           *
                        ;**********************************
0000                            .text
0000      40340003              mov #03h, R4

                        ;**********************************
                        ;*  Reserve 1 byte in var1        *
                        ;**********************************
0000                    ptr     .usect  "var1",1

                        ;**********************************
                        ;*  Reserve 100 more bytes in var1  *
                        ;**********************************
0001                    array   .usect  "var1",100

0004      50340037              add #37h, R4 ; still in .text

                        ;**********************************
                        ;*  Reserve 50 more bytes in var1  *
                        ;**********************************
0065                    dflag   .usect  "var1",50

0008     -4014005b              mov dflag, R4; still in .text

                        ;**********************************
                        ;*  Reserve 100 bytes in var2     *
                        ;**********************************
0000                    vec     .usect  "var2", 100

000c     -90140000              cmp vec, R4  ; still in .text

                        ;**********************************
                        ;*  Declare external .usect symbol  *
                        ;**********************************
                                .global array
```



Example 4.8: The .usect Directive

Syntax **.word** *value$_1$* [, ... , *value$_n$*]

Description The .word directive places one or more 16–bit values into consecutive two–byte pairs in the current section.

The *values* can be either absolute or relocatable expressions. 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. If you use a label, it points to the first word that is initialized.

Note that when you use .word in a .struct/.endstruct sequence, it defines a member's size; it does not initialize memory.

Example This example shows how to use the .word directive to initialize words. The symbol WordX points to the first word that is reserved.

```
1 0000   0c80  Wordx:  .word   3200, 1+'B', -0afh, 'X'
  0002   0043
  0004   ff51
  0006   0058
```

Topics

Tables

Notes

5 Instruction Set Summary

This chapter summarizes the MSP430 family instruction set.

5.1 Symbols and Abbreviations

The following table lists the instruction set symbols and abbreviations used throughout the rest of this chapter.

Symbol	Definition	Symbol	Definition
src	The source operand defined by As and S-reg	dst	The destination operand defined by Ad and D-reg
As	The bits representing the addressing mode used for the source	Ad	The bit representing the addressing mode used for the destination
S-reg	The used Working Register for the source src	D-reg	The used Working Register for the destination dst
R0 or PC	Register 0 or Program Counter	R1 or SP	Register 1 or Stack Pointer
R2 or SR/CG1	Register 2 or Status Register/Constant Generator 1	R3 or CG2	Register 3 or Constant Generator 2
R4 to R15	Working Register, general purpose	Rn	Working Register with n=4-15, general purpose
#	Immediate Data	@	Register indirect addressing
&	Absolute address	-->	Data transfer direction
label	16-bit label	TOS	Top of Stack
C	Carry Bit	N	Negative Bit
V	Overflow Bit	Z	Zero Bit
.B	The suffix .B at the instruction memonic will result in a byte operation	.W	The suffix .W or no suffix at the instruction memonic will result in a word operation
MSB	Most significant Bit	LSB	Least significant Bit

Table 5.1: Symbols and Abbreviations used in the Instruction Set Summary

5.2 Addressing Modes

All seven addressing modes for the source operand and all four addressing modes for the destination operand can address the complete address space. The bit numbers show the contents of the As resp. Ad mode bits.

As	Ad	Addressing Mode	Syntax	Description
00	0	Register Mode	Rn	Register contents are operand
01	1	Indexed Mode	X(Rn)	(Rn + X) points to the operand. X is stored in the next word
01	1	Symbolic Mode	ADDR	(PC + X) points to the operand. X is stored in the next word. Indexed Mode X(PC) is used
01	1	Absolute Mode	&ADDR	The word following the instruction contains the absolute address.
10	-	Indirect Register Mode	@Rn	Rn is used as a pointer to the operand
11	-	Indirect Autoincrement	@Rn+	Rn is used as a pointer to the operand. Rn is incremented afterwards
11	-	Immediate Mode	#N	The word following the instruction contains the immediate constant N. Indirect Autoincrement Mode @PC+ is used

Table 5.2: Addressing Modes

Note: Addressing Modes

The addressing modes using the PC as the working register use the normal effects of the addressing modes. The special addressing modes are caused by the pointing of the PC to the ROM word following the currently executed instruction.

5.3 Instruction Set Summary

				Status Bits			
				V	N	Z	C
*	ADC(.B)	dst	dst + C → dst	x	x	x	x
	ADD(.B)	src,dst	src + dst → dst	x	x	x	x
	ADDC(.B)	src,dst	src + dst + C → dst	x	x	x	x
	AND(.B)	src,dst	src .and. dst → dst	0	x	x	x
	BIC(.B)	src,dst	.not.src .and. dst → dst	-	-	-	-
	BIS(.B)	src,dst	src .or. dst → dst	-	-	-	-
	BIT(.B)	src,dst	src .and. dst	0	x	x	x
*	BR	dst	Branch to	-	-	-	-
	CALL	dst	PC+2 → stack, dst → PC	-	-	-	-
*	CLR(.B)	dst	Clear destination	-	-	-	-
*	CLRC		Clear carry bit	-	-	-	0
*	CLRN		Clear negative bit	-	0	-	-
*	CLRZ		Clear zero bit	-	-	0	-
	CMP(.B)	src,dst	dst - src	x	x	x	x
*	DADC(.B)	dst	dst + C → dst (decimal)	x	x	x	x
	DADD(.B)	src,dst	src + dst + C → dst (decimal)	x	x	x	x
*	DEC(.B)	dst	dst - 1 → dst	x	x	x	x
*	DECD(.B)	dst	dst - 2 → dst	x	x	x	x
*	DINT		Disable interrupt	-	-	-	-
*	EINT		Enable interrupt	-	-	-	-
*	INC(.B)	dst	Increment destination, dst +1 → dst	x	x	x	x
*	INCD(.B)	dst	Double-Increment destination, dst+2→dst	x	x	x	x
*	INV(.B)	dst	Invert destination	x	x	x	x
	JC/JHS	Label	Jump to Label if Carry-bit is set	-	-	-	-
	JEQ/JZ	Label	Jump to Label if Zero-bit is set	-	-	-	-
	JGE	Label	Jump to Label if (N .XOR. V) = 0	-	-	-	-
	JL	Label	Jump to Label if (N .XOR. V) = 1	-	-	-	-
	JMP	Label	Jump to Label unconditionally	-	-	-	-
	JN	Label	Jump to Label if Negative-bit is set	-	-	-	-

Legend:				
	0	Status bit always cleared	1	Status bit always set
	x	Status bit cleared or set on results	-	Status bit not affected
	*	Emulated Instructions		

Table 5.3: MPS430 Family Instruction Set Summary

				Status Bits			
				V	N	Z	C
	JNC/JLO	Label	Jump to Label if Carry-bit is reset	-	-	-	-
	JNE/JNZ	Label	Jump to Label if Zero-bit is reset	-	-	-	-
	MOV(.B)	src,dst	src → dst	-	-	-	-
*	NOP		No operation	-	-	-	-
*	POP(.B)	dst	Item from stack, SP+2 → SP	-	-	-	-
	PUSH(.B)	src	SP - 2 → SP, src → @SP	-	-	-	-
	RETI		Return from interrupt TOS → SR, SP + 2 → SP TOS → PC, SP + 2 → SZP	x	x	x	x
*	RET		Return from subroutine TOS → PC, SP + 2 → SP	-	-	-	-
*	RLA(.B)	dst	Rotate left arithmetically	x	x	x	x
*	RLC(.B)	dst	Rotate left through carry	x	x	x	x
	RRA(.B)	dst	MSB → MSB ....LSB → C	0	x	x	x
	RRC(.B)	dst	C → MSB ....LSB → C	x	x	x	x
*	SBC(.B)	dst	Subtract carry from destination	x	x	x	x
*	SETC		Set carry bit	-	-	-	1
*	SETN		Set negative bit	-	1	-	-
*	SETZ		Set zero bit	-	-	1	-
	SUB(.B)	src,dst	dst + .not.src + 1 → dst	x	x	x	x
	SUBC(.B)	src,dst	dst + .not.src + C → dst	x	x	x	x
	SWPB	dst	swap bytes	-	-	-	-
	SXT	dst	Bit7 → Bit8 .... Bit15	0	x	x	x
*	TST(.B)	dst	Test destination	x	x	x	x
	XOR(.B)	src,dst	src .xor. dst → dst	x	x	x	x

Legend:				
	0	The Status Bit is cleared	1	The Status Bit is set
	x	The Status Bit is affected	-	The Status Bit is not affected
	*	Emulated Instructions		

Table 5.3: MPS430 Family Instruction Set Summary (Concluded)

Note: Emulated Instructions

All marked instructions (*) are emulated instructions. The emulated instructions use core instructions combined with the architecture and implementation of the CPU for higher code efficiency and faster execution.

5.4 Clock cycles, Length of Instruction

The operating speed of the CPU is independent from individual instructions. It depends on the instruction format and the addressing modes. The number of clock cycles refer to the internal oscillator frequency.

5.4.1 Format I Instructions

Address Mode		#of cycles	Length of instruction	Example
As	Ad			
00, Rn	0, Rm	1	1	MOV R5,R8
	0,PC	2	1	BR R9
00, Rn	1, x(Rm)	4	2	ADD R5,3(R6)
	1, EDE		2	XOR R8,EDE
	1, &EDE		2	MOV R5,&EDE
01, x(Rn)	0, Rm	3	2	MOV 2(R5),R7
01, EDE			2	AND EDE,R6
01, &EDE				MOV &EDE,R8
01, x(Rn)	1, x(Rm)	6	3	ADD 3(R4),6(R9)
01, EDE	1, TONI		3	CMP EDE,TONI
01, &EDE	1, &TONI		3	MOV 2(R5),&TONI
				ADD EDE,&TONI
10, @Rn	0, Rm	2	1	AND @R4,R5
10, @Rn	1, x(Rm)	5	2	XOR @R5,8(R6)
	1, EDE		2	MOV @R5,EDE
	1, &EDE		2	XOR @R5,&EDE
11, @Rn+	0, Rm	2	1	ADD @R5+,R6
	0, PC	3	1	BR @R9+
11, #N	0, Rm	2	2	MOV #20,R9
	0, PC	2	2	BR #2AEh
11, @Rn+	1, x(Rm)	5	2	MOV @R9+,2(R4)
11, #N	1, EDE		3	ADD #33,EDE
11, @Rn+	1, &EDE		2	MOV @R9+,&EDE
11, #N			3	ADD #33,&EDE

Table 5.4: Format I Instructions

Note: Cycle Time of the DADD Instruction

The DADD instruction needs 1 extra cycle.

5.4.2 Format II Instructions

Address Mode $A_{(s/d)}$	#of cycles		Length of instruction [words]	Example
	RRC RRA SWPB SXT	PUSH/ CALL		
00, Rn	1	3/4	1	SWPB R5
01, x(Rn)	4	5	2	CALL 2(R7)
01, EDE	4	5	2	PUSH EDE
10, @Rn	3	4	1	RRC @R9
11, @Rn+ see Note	3	4/5	1	SWPB @R10+
11, #N	3	4/5	2	CALL #81h

Table 5.5: Format II Instructions

Note: Immediate mode in destination field

Instructions should not use immediate mode in the destination field. This would result in unpredictable program operation.

5.4.3 Format III Instructions

Jxx - instructions need all the same #-of-cycles independent of a successfull Jump or not.

Clock Cycle:	2 Cycle.
Length of Instruction:	1 word.

5.4.4 Miscellanous Instructions or Operators

RETI	Clock Cycle:	5 Cycle.
	Length of instruction:	1 word.
Interrupt	Clock Cycle:	6 Cycle.

Topics

Examples

6 Macro Language

The 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 enables you to:

- 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.

6.1 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 to 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. In this chapter, we use the terms *macro parameters* and *substitution symbols* interchangeably.

Using a macro is a three–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:

- Macros can be defined at the beginning of a **source file** or in an .include/.copy file.
- 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.

Step 2: **Call the macro.** After you have defined a macro, you can call it by using the macro name as an opcode 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 .mnolist directive.

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 encountered macro. This allows you to expand the functions of directives and instructions, as well as add new instructions.

6.2 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 an .include/.copy file; they can also be defined in a macro library.

The contents of a macro definition must be contained in the same file. Macro definitions can be nested, and they can call other macros.

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

macname **.macro** *[parameter$_1$] [,parameter$_2$] ... [,parameter$_n$]*

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 32 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 a directive that identifies the source statement as the first line of a macro definition. You must place .macro in the opcode field.
[parameters]	are optional substitution symbols that appear as operands for the .macro directive.
model statements	are instructions or assembler directives that are executed each time the macro is called.
macro directives	are used to control macro expansion.
[.mexit]	functions as a “goto .endm". The .mexit directive is useful when error testing confirms that macro expansion will fail.
.endm	terminates the macro definition.

Macro definition: The following code defines a macro, add3, with 3 parameters:

```
        1                        ;* add3  arg1, arg2, arg3
        2                        ;*       arg3 = arg1 + arg2 + arg3
        3
        4                        add3     .macro arg1, arg2, arg3
        5
        6                                 mov arg1, R4
        7                                 mov R4, R5
        8                                 mov arg2, R4
        9                                 add R4, R5
       10                                 mov arg3, R4
       11                                 add R5, R4
       12                                 mov R4, arg3
       13                                 .endm
```

Macro Call: The following code calls the add3 macro with 3 arguments

```
       14
       15 0000                            add3 x, y ,z
```

Macro Expansion: The following code shows the substitution of the macro definition for the macro call. The assembler passes the arguments (supplied in the macro call) by variable to the parameters (substitution symbols).

```
1
1         0000  -40140000                 mov x, R4
1         0004   4405                     mov R4, R5
1         0006  -4014fff9                 mov y, R4
1         000a   5405                     add R4, R5
1         000c  -4014fff4                 mov z, R4
1         0010   5504                     add R5, R4
1         0012  -4480ffee                 mov R4, z

       16
       17
       18                        ;* Reserve space for vars
       19 0000                            .bss x,1
       20 0001                            .bss y,1
       21 0002                            .bss z,1
```

Example 6.1: Macro Definition, Call, and Expansion

If you want to include comments with your macro definition but *don't* want those comments to appear in the macro expansion, precede your comments with an exclamation point. If you *do* want your comments to appear in the macro expansion, use an asterisk or semicolon in place of the exclamation point.

6.3 Macro Parameters/Substitution Symbols

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

Substitution Symbols

Macro parameters are substitution symbols. Substitution symbols are symbols that represent a character string. Besides being used as macro parameters, these symbols can also be used outside of macros to equate a character string to a symbol name.

Valid substitution symbols may be 32 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.

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 the arguments with quotation marks.

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

Macro Definition

```
Parms     .macro a,b,c
;            a = :a:
;            b = :b:
;            c = :c:
          .endm
```

Calling the Macro, Parms

```
          Parms  100,label                   Parms  100,label,x,y
;            a = 100                                ;  a = 100
;            b = label                              ;  b = label
;            c = " "                                ;  c = x,y

          Parms  100,x                        Parms  "100,200,300",x,y
;            a = 100                                ;  a = 100,200,300
;            b = " "                                ;  b = x
;            c = x                                  ;  c = y

          Parms  """string""",x,y
;            a = "string"
;            b = x
;            c = y
```

Example 6.2: Calling a Macro With Varying Numbers of Arguments

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.

  The syntax of the .asg directive is:

  .asg *["] character string ["], 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 example shows character strings being assigned to substitution symbols.

```
          .asg  R13,FP                    ;  frame pointer
          .asg  @  ,Ind                   ;  indirect addressing
          .asg  """string""",strng        ;  string
          .asg  "a,b,c",parms             ;  parameters
          mov   Ind(FP),R4                ;  mov @(R13),R4
```

Example 6.3: Using the .asg Directive

- The **.eval** directive performs arithmetic on numeric substitution symbols.

  The syntax of the .eval directive is:

  .eval *well–defined expression, substitution symbol*

  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 example shows arithmetic being performed on substitution symbols.

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

Example 6.4: Using the .eval Directive

In the example, 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. If you want to calculate a *value* from an expression, however, you must use the .eval directive.

The .asg directive only assigns a character string to a substitution symbol, while the .eval directive evaluates an expression and then assigns the character string equivalent to a substitution symbol.

Built-In Substitution Symbol Functions

The following built–in substitution symbol functions enable you to make decisions based on 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 to these functions are substitution symbols or character–string constants.

In the following function definitions, *a* and *b* are parameters that represent substitution symbols or character string constants. The term *string*, used below, refers to the string value of the parameter.

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

```
         .asg    Label, x                         ; x = label
         .if     ($symcmp(x,"Label") = 0)         ; evaluates to true
         cmp     x,R4
         .endif
         .asg    "L1,L2,L3", list                 ; list = L1,L2,L3
         .if     ($ismember(x, list))             ; x = L1  list = L2,L3
         cmp     x,R4
         .endif
```

Example 6.5: Using Built–In Substitution Symbol Functions

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 the example, the y is substituted for x; z is substituted for y; and x is substituted for z. The assembler recognizes this as infinite recursion and ceases 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"
        cmp    x, R4

;*      cmp    x, R4        ; recursive expansion
```

Example 6.6: Recursive Substitution

Forcing Substitutions

In some cases, substitution symbols are not recognizable to the assembler. The forced substitution operator, which is a set of colons, enables you to force the substitution of a symbol's character string. Simply surround 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 it expands any other substitution symbols.

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

```
force    .macro   x
         .asg     0, x
         .loop 8               ; loop/.endloop are discussed
;*                               in Section 4.0
AUX:x:   .equ x                ; The x in AUXx would not be
;*                               recognizable as a substituion
;*                               symbol by the assembler
         .eval  x+1, x
         .endloop
         .endm

         force    ;macro call
```

This would generate the following source code:

```
         AUX0    .equ 0
         AUX1    .equ 1
               .
               .
         AUx7    .equ 7
```

Example 6.7: Using the Forced Substitution Operator

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 expression$_1$*, *well–defined expression$_2$*):

  In this method, expression$_1$ represents the substring's starting position, and expression$_2$ 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.*

The next two examples show built–in substitution symbol functions used with subscripted substitution symbols.

In the first example, subscripted substitution symbols redefine the pop instruction so that it handles indirect addressing.

```
popx      macro    x
         .var      tmp,len
         .asg      :x(1):,tmp
         .if       $symcmp(tmp,"@") = 0
         .eval     $symlen(x),len
         .asg      :x(2,len):,tmp
         .if       $isreg(tmp) = 1
         pop       0(tmp)
         .else
         .emsg     "Bad Register Name"
         .endif
         .else
         .emsg     "Bad Operand"
         .endif
         .endm

          popx     @R4         ; macro call
```

Example 6.8: Using Subscripted Substitution Symbols to Redefine an Instruction

In the second example, 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.

```
substr   .macro    start,strg1,strg2,pos
         .var      len1,len2,i,tmp
         .if       $symlen(start) = 0
         .eval     1,start
         .endif
         .eval     0,pos
         .eval     start,i
         .eval     $symlen(strg1),len1
         .eval     $symlen(strg2),len2
         .loop
         .break    i = (len2-len1+1)
         .asg      ":strg2(i,len1):",tmp
         .if       $symcmp(strg1,tmp) = 0
         .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
```

Example 6.9: Using Subscripted Substitution Symbols to Find Substrings

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, after expansion, these symbols are lost.

.var *sym_1 [,sym_2] ... [,sym_n]*

The .var directive is used in the last two examples.

6.4 Macro Libraries

One of the ways you can define macros is in 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.

.mlib *macro library filename*

When the assembler encounters an .mlib directive, it opens the library and creates a table of its 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 manner as other macros. You can control the listing of library entry expansions with the .mlist directive. Only macros that are actually called from the library are extracted, and they are extracted only once.

You can create a macro library with the archiver by simply 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.

6.5 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*

code block to execute when the .if expression is true (nonzero)

[.elseif *well–defined expression***]**

code block to execute when the .elseif expression is true (nonzero)

[.else]

code block to execute when the .elseif expression is false (zero)

.endif

The .elseif and .else directives are optional, and they can be used more than once within a conditional assembly code block. When they are omitted, and when the .if expression is false (zero), the assembler continues to the code following the .endif directive.

The **.loop/.break/.endloop** directives enable you to assemble a code block repeatedly. The format of a repeatable block is:

.loop *[well–defined expression]*

code block to repeatedly assemble

[.break *[well–defined expression]***]**

continue to repeatedly assemble when the .break expression is false (zero)

.endloop

code block to execute when the .break expression is true (nonzero) or when the .break expression is omitted and the loop count equals *expression.*

The .loop directive's optional expression evaluates to the loop count. If the expression is omitted, the loop count defaults to 1024, unless the assembler encounters a .break directive.

The .break directive and its expression are optional. 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 and the loop count equals *expression.* When the loop is broken, the assembler continues with the code after the .endloop directive.

The next three examples show the .loop/.break/.endloop directives, properly nested conditional assembly directives, and built–in substitution symbol functions used in a conditional assembly code block.

```
          .asg     1,x
          .loop

          .break   (x==10)    ; If x==10, quit loop/break with expression

          .eval    x+1,x
          .endloop
```

Example 6.10: The .loop/.break/.endloop Directives

```
          .asg     1,x
          .loop

          .if      (x==10)    ; If x==10, quit loop
          .break              ; force break
          .endif

          .eval    x+1,x
          .endloop
```

Example 6.11: Nested Conditional Assembly Directives

6.6 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 the label with a question mark, and the assembler will replace the question mark with a period followed by a unique number. When the macro is expanded, *you will not see the unique number in the listing file.* Your label will appear with the question mark as it did in the macro definition. The syntax for a unique label is:

*label***?**

```
        1                         min       .macro x,y
        2                                   mov x, R4
        3                                   cmp y, r4
        4                                   jl m1?
        5                                   mov y, R4
        6                         m1?
        7                                   .endm
        8
        9
       10
       11 0000                              .bss var1
       12 0001                              .bss var2
       13 0002                              .bss var3
       14
       15 0000                              min var1, var2
1         0000  -40140000                   mov var1, R4
1         0004  -9014fffb                   cmp var2, r4
1         0008  '3802                       jl m1?
1         000a  -4014fff5                   mov var2, R4
1         000e                    m1?
       16
       17 000e                              min var2, var3
1         000e  -4014fff1                   mov var2, R4
1         0012  -9014ffee                   cmp var3, r4
1         0016  '3802                       jl m1?
1         0018  -4014ffe8                   mov var3, R4
1         001c                    m1?
       18


LABEL                            VALUE        DEFN     REF

m1.1                             000e    '     15       15
m1.2                             001c    '     17       17
var1                             0000    -     11       15
var2                             0001    -     12       15       15       17
var3                             0002    -     13       17       17
```

Example 6.12: Unique Labels 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 30 characters. If the macro is expanded from 10 to 999 times, the maximum label length is 29. The label with its unique suffix is shown in the cross–listing file.

6.7 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 way the assembler does, incrementing the error count and preventing the assembler from producing an object file.

.wmsg sends warning messages to the listing file. The .wmsg directive functions in the same manner as the .emsg directive but increments the warning count and does not prevent the generation of an object file.

.mmsg sends warnings or 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 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.

The example shows user messages in macros and macro comments that will not appear in the macro expansion.

```
TEST     .MACRO    x,y
;
;  This macro checks for the correct number of parameters.
;  The macro generates an error message if x an y are not present.
;
         .if       ($symlen(x) == 0|$symlen(y) == 0))  ; Test for
                                                       ; proper input
         .emsg     "ERROR - missing parameter in call to TEST"
         .mexit
         .else
           .
           .
         .endif
         .if
           .
           .
         .endif
         .endm

   1 error, no warnings
```

Example 6.13: Producing Messages in a Macro

6.8 Formatting 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. The assembler provides three sets of directives that 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 to the listing all code encountered in those blocks. *By default, the assembler behaves as if you had used .mlist.*
.mnolist	suppresses the listing expansion of macros and .loop/.endloop blocks.

- **False Conditional Block Listing**

.fclist	causes the assembler to print to the listing file all false conditional blocks that do not generate code. Conditional blocks appear in the listing exactly as they appear in the source code. *By default, the assembler behaves as if you had used .fclist.*
.fcnolist	suppresses the listing of false conditional blocks. Only the code in conditional blocks that actually assembles appears in the listing. The .if, .elsif, .else, and .endif directives do not appear in the listing.

- **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. *By default, the assembler behaves as if you had used .ssnolist.*

6.9 Using Recursive and Nested Macros

The macro language supports recursive and nested macro calls. This means that you can call other macros from inside a macro definition. When you use nested macros, you can call different macros from your 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.

The following example shows nested macros. Note that 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.

```
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 6.14: Using Nested Macros

The next example shows recursive 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 the A register. The fact macro accomplishes this by calling fact1, which calls itself recursively.

```
         .fcnolist

fact1    .macro n

         .if n == 1
             mov #globcnt, R4             ; leave answer in R4 reg.

         .else
             .eval n-1, temp              ; compute decrement of n
             .eval globcnt*temp, globcnt  ; multiply to get new result
             fact1 temp                   ; recursive call

         .endif
         .endm

fact     .macro n

         .if ! $iscons(n)                 ; type check input
             .emsg "Parm not a constant"

         .elseif n < 1                    ; type check input
             mov #0, R4

         .else
             .var temp
             .asg n, globcnt

             fact1 n                      ; perform recursive procedute

         .endif
         .endm
```

Example 6.15: Using Recursive Macros

6.10 Macro Directives Summary

Creating Macros

Mnemonic and Syntax	Description
macname **.macro** *[parameter$_1$]...[parameter$_n$]*	Define macro
.mlib *filename*	Identify library containing macro definitions
.mexit	Go to .endm
.endm	End macro definition

Manipulating Substitution Symbols

Mnemonic and Syntax	Description
.asg *["]character string["], substitution symbol*	Assign character string to substitution symbol
.eval *well–defined expression, substitution symbol*	Perform arithmetic on numeric substitution symbols
.var *substitution symbol$_1$...[substitution symbol$_n$]*	Define local macro symbols

Conditional Assembly

Mnemonic and Syntax	Description
.if *well–defined expression*	Begin conditional assembly
.elseif *well–defined expression*	Optional conditional assembly block
.else	Optional conditional assembly block
.endif	End conditional assembly
.loop *[well–defined expression]*	Begin repeatable block assembly
.break *[well–defined expression]*	Optional repeatable block assembly
.endloop	End repeatable block assembly

Producing Assembly–Time Messages

Mnemonic and Syntax	Description
.emsg	Send error message to standard output
.wmsg	Send warning message to standard output
.mmsg	Send assembly–time message to standard output

Formatting the Listing

Mnemonic and Syntax	Description
.fclist	Allow false conditional code block listing (default)
.fcnolist	Inhibit false conditional code block listing
.mlist	Allow macro listings (default)
.mnolist	Inhibit macro listings
.sslist	Allow expanded substitution symbol listing
.ssnolist	Inhibit expanded substitution symbol listing (default)

Topics

Figures

Notes

7 Archiver Description

The MSP430 family archiver lets you combine several individual files into a single file called an **archive** or a **library**. Each file within the archive is called a **member.** Once you have created an archive, you can use the archiver to add more files to the library, delete or replace existing members, or extract members.

You can build libraries out of 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 to build a library of object modules. For example, you could write several arithmetic routines, assemble them, and then use the archiver to collect the object files into a single, logical group. You can then specify an object library as linker input. The linker will search through the library and include any members that resolve external references.

You can also use the archiver to build macro libraries. You can create several separate source files, each of which contains a single macro, and then use the archiver to collect these macros into a single, functional group. The .mlib assembler directive lets you specify the name of a macro library to the assembler; during the assembly process, the assembler will search the specified library for the macros that you call.

7.1 Archiver Development Flow

The figure shows the archiver's role in the assembly language development process. Both the assembler and the linker accept libraries as input.

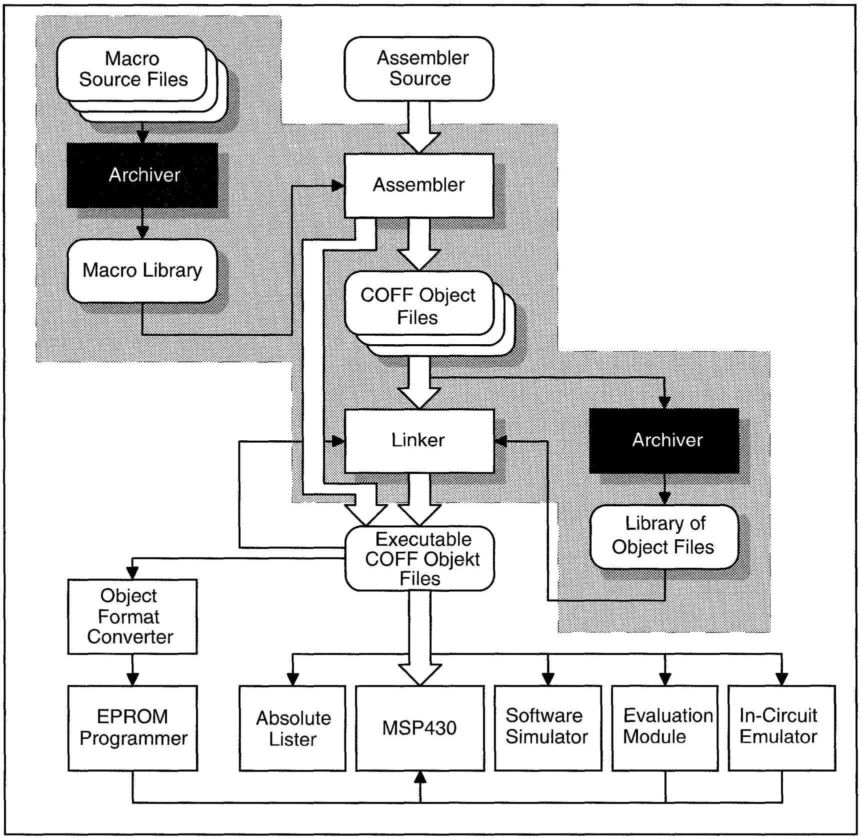


Figure 7.1: Archiver Development Flow

7.2 Invoking the Archiver

To invoke the archiver, enter:

ar430 *[-]command[option] libname [filename$_1$ ... filename$_n$]*

ar430 is the command that invokes the archiver.

libname names an archive library. If you don't specify an extension for *libname,* the archiver uses the default extension **.lib**.

filename names individual member files that are associated with the library. If you don't specify an extension for a *filename,* the archiver uses the default extension **.obj**.

command tells the archiver how to manipulate the members in the library. 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. These are valid archiver commands:

- **-a** adds the specified files to the library. Note that 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. It is possible to have several members with the same name in an archive. If you want to *replace* existing members, use the **r** command.
- **-d** deletes the specified members from the library.
- **-r** replaces the specified members in the library. If you don't specify any 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 don't specify any filenames, the archiver lists all the members in the specified library.
- **-x** extracts the specified files. If you don't specify any member names, the archiver extracts all the members in the library. When the archiver extracts a member, it simply copies the member into the current directory; it *doesn't* remove it from the library.

In addition to one of the *commands,* you can specify the following *options*:

- **-e** tells the archiver not to use the default extension .obj for member names. This allows the use of filenames without extensions.
- **-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.)
- **-v** (verbose) provides a file–by–file description of the creation of a new library from an old library and its constituent members.

Note: Naming Library Members

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, and the library contains more than one member with the specified name, the archiver deletes, replaces, or extracts the first member with that name.

7.3 Archiver Examples

Here are some examples of using the archiver.

- **Example 1**

  This example creates a library called function.lib that contains the files sine.obj, cos.obj, and flt.obj.

```
ar430 -a function sine cos flt
MSP430 Archiver                        Version 1.00
Copyright (c) 1994 Texas Instruments Incorporated
    ==>    new archive 'function.lib'
    ==>    building archive 'function.lib'
```

  Because these examples use the default extensions (.lib for the library and .obj for the members), it is not necessary to specify them.

- **Example 2**

  You can print a table of contents of function.lib with the -t option:

```
ar430 -t function
MSP430 Archiver                        Version 1.00
Copyright (c) 1994 Texas Instruments Incorporated
      FILE NAME    SIZE   DATE
-----------------  -----  -----------------------
      sine.obj     248    Mon Feb 14 01:25:44 1994
      cos.obj      248    Mon Feb 14 01:25:44 1994
      flt.obj      248    Mon Feb 14 01:25:44 1994
```

- **Example 3**

  You can explicitly specify extensions if you don't want the archiver to use the default extensions; for example:

```
ar430 -av function.fn sine.asm cos.asm flt.asm
MSP430 Archiver                        Version 1.00
Copyright (c) 1994 Texas Instruments Incorporated
    ==>     add 'sine.asm'
    ==>     add 'cos.asm'
    ==>     add 'flt.asm'
    ==>     building archive 'function.fn'
```

  This creates a library called function.fn that contains the files sine.asm, cos.asm, and flt.asm. (-v is the verbose option.)

- **Example 4**

  If you want to add new members to the library, specify

```
ar430 -as function tan.obj arctan.obj area.obj
MSP430 Archiver                        Version 1.00
Copyright (c) 1994 Texas Instruments Incorporated
    ==>     symbol defined: 'R2D2'
    ==>     symbol defined: 'Christmas'
    ==>     building archive 'function.lib'
```

  Because this example doesn't specify an extension for the libname, the archiver adds the files to the library called function.lib. If function.lib didn't exist, the archiver would create it. (The -s option tells the archiver to list the global symbols that are defined in the library.)

- **Example 5**

  If you want to modify a member in a library, you can extract it, edit it, and replace it. In this example, assume there's 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 doesn't remove push.asm from the library, though. Now you can edit the extracted file. To replace the copy of push.asm that's in the library with the edited copy, enter:

  ```
  ar430 -r macros push.asm
  ```

Topics

Examples

Figures

Tables

Notes

8 Linker Description

The MSP430 family linker creates executable modules by combining COFF object files. The concept of COFF *sections* is basic to linker operation.

As the linker combines object files, it:

- 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 and provides two powerful directives, MEMORY and SECTIONS, that allow you to:

- define a memory model that conforms to target system memory
- combine object file sections
- allocate sections into specific areas of memory
- define or redefine global symbols at link time

8.1 Linker Development Flow

The following figure illustrates the linker's role in the assembly language 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 COFF object module that can be downloaded to one of several development tools or executed by a MSP430 device.

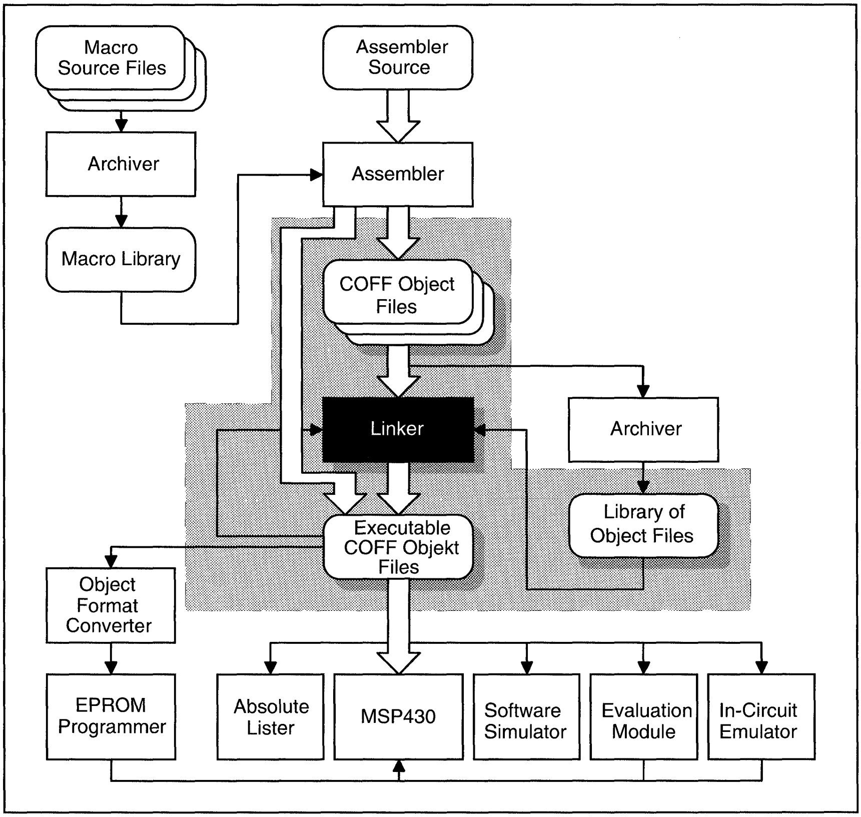


Figure 8.1: Linker Development Flow

8.2 Invoking the Linker

The general syntax for invoking the linker is:

lnk430 [-option] *filename*$_1$ ... *filename*$_n$

lnk430 is the command that invokes the linker.

options

can appear anywhere on the command line or in a linker command file. (Options are discussed in Section 8.3.)

filenames

can be object files, linker command files, or archive libraries. The default extension for all input files is ***.obj***; any other extension must be explicitly specified. The linker can determine whether the input file is an object file or an ASCII file that contains linker commands. The default output filename is ***a.out***.

There are three 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.

  ```
  lnk430 file1.obj file2.obj -o link.out
  ```

- Enter the **lnk430** command with no filenames and no options; the linker will prompt for them:

  ```
  Command files :
  Object files [.obj] :
  Output files [ ] :
  Options :
  ```

  For *command files,* enter one or more command file names.

  For *object files,* enter one or more object file names. The default extension is *.obj.* Separate the filenames with spaces or commas; if the last character is a comma, the linker will prompt for an additional line of object file names.

  The *output file* is the name of the linker output module. This overrides any -o options entered with any of the other prompts. If there are no -o options and you do not answer this prompt, the linker will create an object file with a default filename of ***a.out***.

  The *options* prompt is for additional options, although you can also enter them in a command file. Enter them with hyphens, just as you would on the command line.

- Put filenames and options in a linker command file. For example, assume the file linker.cmd contains the following lines:

  ```
  -o link.out
  file1.obj
  file2.obj
  ```

  Now you can invoke the linker from the command line; specify the command file name as an input file:

```
lnk430 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:

```
lnk430 -m link.map linker.cmd file3.obj
```

The linker reads and processes a command file as soon as it encounters it 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.

8.3 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 (-). The order in which options are specified is unimportant, except for the -l and -i options. Options are separated from arguments (if they have them) by an optional space.

Option	Description
-a	Produce an absolute, executable module. This is the default; if neither -a nor -r is specified, the linker acts as if -a is specified.
-ar	Produce a relocatable, executable object module.
-e *global symbol*	Define a *global symbol* that specifies the primary entry point for the output module.
-f *fill value*	Set the default fill value for holes within output sections; *fill value* is a 16–bit constant.
-h	Make all global symbols static.
-i *dir* †	Alter the library–search algorithm to look in *dir* before looking in the default location. This option must appear before the -l option.
-l *filename* †	Name an archive library file as linker input; *filename* is an archive library name.
-m *filename* †	Produce a map or listing of the input and output sections, including holes, and place the listing in *filename.*
-o *filename* †	Name the executable output module. The default filename is a*.out.*
-q	Request a quiet run (suppress the banner).
-r	Retain relocation entries in the output module.
-s	Strip symbol table information and line number entries from the output module.
-u *symbol*	Place an unresolved external *symbol* into the output module's symbol table.
-x	Force rereading of libraries. Resolves “back" references.
-z *filename* †	Produce an additional byte formatted ASCII file loadable by the evaluation module. The default filename is the output filename with the extension .txt.

† The *filename* must follow operating system conventions.

Table 8.1: Linker Options Summary

8.3.1 Relocation Capabilities (-a and -r Options)

One of the tasks the linker performs is *relocation.* Relocation is the process of adjusting all references to a symbol when the symbol's address changes. The linker supports two options (-a and -r) that allow you to produce an absolute or a relocatable output module. Default is -a.

- **Producing an Absolute Output Module (-a Option)**

  When you use the **-a** option without the -r 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
  - an optional header that describes information such as the program entry point
  - *no* unresolved references

  This example links file1.obj and file2.obj and creates an absolute output module called a.out:

  ```
  lnk430 -a file1.obj file2.obj
  ```

- **Producing a Relocatable Output Module (-r Option)**

  When you use the **-r** option without the -a option, the linker retains relocation entries in the output module. If the output module will be relocated (at load time) or relinked (by another linker execution), use -r to retain the relocation entries.

  The linker produces an *unexecutable* file when you use the -r option without -a. A file that is not executable does not contain special linker symbols or an optional header. The file may 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:

  ```
  lnk430 -r 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)

- **Producing an *Executable* Relocatable Output Module (-ar)**

  If you invoke the linker with both the -a and -r 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, but the relocation information is retained.

  This example links file1.obj and file2.obj and creates an executable, relocatable output module called xr.out:

  ```
  lnk430 -ar file1.obj file2.obj -o xr.out
  ```

  Note that you can string the options together (lnk430 -ar) or you can enter them separately (lnk430 -a -r).

- **Relocating or Relinking an Absolute Output Module**

  The linker issues a warning message (but continues executing) when it encounters a file that contains no relocation or symbol table information. 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).

8.3.2 Define an Entry Point (-e *global symbol* Option)

The memory address that a program begins executing from is called the **entry point.** When a loader loads a program into target memory, the program counter must be initialized to the entry point; the PC then points to the beginning of the program.

The linker can assign one of four possible 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. Possible entry point values include:

- The value specified by the -e option. The syntax is **-e** *global symbol* where *global symbol* defines the entry point and must appear as an external symbol in one of the input files.
- Zero (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.

```
lnk430 -e begin file1.obj file2.obj
```

8.3.3 Set Default Fill Value (-f *cc* Option)

The -f option fills the holes formed within output sections or initializes uninitialized sections when they are combined with initialized sections. This allows you to initialize memory areas during link time without reassembling a source file. The argument *cc* is a 16–bit constant (up to four hexadecimal digits). If you do not use -f, the linker uses 0 as the default fill value.

This example fills holes with the hexadecimal value ABCD:

```
lnk430 -f 0ABCDh file1.obj file2.obj
```

8.3.4 Make All Global Symbols Static (-h Option)

The -h option makes output global symbols static. This is useful when you are using partial linking to link related object files into self–contained modules, then relinking the modules together into a final system. If there are global symbols in one module that have the same name as global symbols in other modules, but you want to treat them as separate symbols, use the -h option when building the modules. The global symbols in the modules, which would normally be visible to the other modules and cause possible redefinition problems in the final link, are made static so they are not visible to the other modules.

For example, assume b1.obj, b2.obj, and b3.obj are related and reference a global variable GLOB. Also assume that d1.obj, d2.obj, and d3.obj are related and reference a separate global variable GLOB. You can link the related files together with the following commands:

```
lnk430 -h -r b1.obj b2.obj b3.obj -o bpart.out
lnk430 -h -r d1.obj d2.obj d3.obj -o dpart.out
```

The -h option guarantees that bpart.out and dpart.out will not have global symbols and therefore two distinct versions of GLOB exist. The -r option is used to allow bpart.out and dpart.out to retain their relocation entries. These two partially linked files can then be linked together safely with the following command:

```
lnk430 bpart.out dpart.out -o system.out
```

8.3.5 Alter the Library Search Algorithm (-i *dir* Option/C_DIR)

Usually, when you want to specify a library as linker input, you simply enter the library name as you would any other input filename; the linker looks for the library 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:

```
lnk430 file1.obj object.lib
```

If you want to use a library that is not in the current directory, use the -l (lowercase "L") linker option. The syntax for this option is **-l** *filename*. The *filename* is the name of an archive library; the space between -l and the filename is optional.

You can augment the linker's directory search algorithm by using the -i linker option or the environment variable. The linker searches for object libraries in the following order:

1) It searches directories named with the -i linker option.
2) It searches directories named with the environment variable C_DIR.
3) If C_DIR is not set, it searches directories named with the assembler's environment variable, A_DIR.
4) It searches the current directory.

-i Linker Options

The -i linker option names an alternate directory that contains object libraries. The syntax for this option is **-i** *dir. dir* names a directory that contains object libraries; the space between -i and the directory name is optional. When the linker is searching for object libraries named with the -l option, it searches through directories named with -i first. Each -i option specifies only one directory, but you can use several -i options per invocation. When you use the -i option to name an alternate directory, it must precede the -l option on the command line or in a command file.

As an example, assume that there are two archive libraries called r.lib and lib2.lib. 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.

	Pathname	Invocation Command
DOS	\ld and \ld2	lnk430 f1.obj f2.obj -i\ld -i\ld2 -lr.lib -llib2.lib

Environment Variable (C_DIR)

An environment variable is a system symbol that you define and assign a string to. The linker uses an environment variable named **C_DIR** to name alternate directories that contain object libraries. The command for assigning the environment variable is:

DOS `set` `C_DIR=`pathname;another pathname ...

The *pathnames* are directories that contain object libraries. Use the -l option on the command line or in a command file to tell the linker which libraries to search for.

As an example, assume that two archive libraries called r.lib and lib2.lib reside in ld and ld2 directories. The table below shows the directories that r.lib and lib2.lib reside in, how to set the environment variable, and how to use both libraries during a link.

	Pathname	Invocation Command

DOS	`\ld and \ld2`	`set C_DIR=\ld;\ld2`
		`lnk430 f1.obj f2.obj -l r.lib -l lib2.lib`

Note that the environment variable remains set until you reboot the system or reset the variable by entering:

DOS `set` `C_DIR=`

The assembler uses an environment variable named **A_DIR** to name alternate directories that contain copy/include files or macro libraries. If C_DIR is not set, the linker will search for object libraries in the directories named with A_DIR.

8.3.6 Create a Map File (-m *filename* Option)

The -m option creates a link map listing and puts it in *filename.* This map describes:

- Memory configuration.
- Input and output section allocation.
- The addresses of external symbols after they have been relocated.

The map file contains the name of the output module and the entry point; it may also contain up to three tables:

- A table showing the new memory configuration **if** any nondefault memory is specified.
- A table showing the linked addresses of each output section and the input sections that make up the output sections.
- A table showing each external symbol and its address. This table has two columns: the left column contains the symbols sorted by name, and the right column contains the symbols sorted by address.

This example links file1.obj and file2.obj and creates a map file called map.out:

```
lnk430 file1.obj file2.obj -m map.out
```

8.3.7 Name an Output Module (-o *filename* 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 -o option. The *filename* is the new output module name.

This example links file1.obj and file2.obj and creates an output module named run.out:

```
lnk430 -o run.out file1.obj file2.obj
```

8.3.8 Specify a Quiet Run (-q Option)

The -q option suppresses the linker's banner when -q is the first option on the command line or in a command file. This option is useful for batch operation.

8.3.9 Strip Symbolic Information (-s Option)

The -s option creates a smaller output module by omitting symbol table information and line number entries. The -s option is useful for production applications when you must create the smallest possible output module.

This example links file1.obj and file2.obj and creates an output module, stripped off line numbers and symbol table information, named nosym.out:

```
lnk430 -o nosym.out -s file1.obj file2.obj
```

Note that using the -s option limits later use of a symbolic debugger and may prevent a file from being relinked.

8.3.5 Introduce an Unresolved Symbol (-u *symbol* Option)

The -u option introduces an unresolved symbol into the linker's symbol table. This forces the linker to search through a library and include the member that defines the symbol. Note that the linker must encounter the -u option *before* it links in the member that defines the symbol.

For example, suppose a library named rts.lib contains a member that defines the symbol symtab; none of the object files you are linking reference symtab. Suppose you plan to relink the output module, however, and you would like to include the library member that defines symtab in this link. Using the -u option as shown below forces the linker to search rts.lib for the member that defines symtab and to link in the member.

```
lnk430 -u symtab file1.obj file2.obj rts.lib
```

If you do not use -u, this member is not included, because there is no explicit reference to it in file1.obj or file2.obj.

8.3.6 Exhaustively Read Libraries (-x option)

The linker normally reads input files, archive libraries included, 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 (this is called a *back reference*), the reference will not be resolved.

You can force the linker to repeatedly reread all libraries with the -x option. The linker will continue to reread libraries until no more references can be resolved. 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:

```
lnk430 -la.lib -lb.lib -la.lib
```

or you can force the linker to do it for you:

```
lnk430 -x -la.lib -lb.lib
```

Linking with the -x option may be slower, so you should use the option only as needed.

8.4 Command Files

Linker command files allow you to put linking information in a file; this is useful when you often invoke the linker 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. Command files are ASCII files that contain one or more of the following:

- Input file names, 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. The SECTIONS directive controls how sections are built and allocated.
- Assignment statements, which define and assign values to global symbols.

To invoke the linker with a command file, enter the **lnk430** command and follow it with the name of the command file:

lnk430 *command file name*

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.

The example shows a sample linker command file called link.cmd.

```
/*******************************************************************/
/*                     Sample Linker Command File                  */
/*******************************************************************/
a.obj                   /*  First input filename                  */
b.obj                   /*  Second input filename                 */
-o prog.out             /*  Option to specify output file         */
-m prog.map             /*  Option to specify map file            */
```

Example 8.1: Linker Command File

This sample file contains only filenames and options. (Note that you can place comments in a command file by delimiting them with /* and */.) To invoke the linker with this command file, enter:

```
lnk430 link.cmd
```

You can place other parameters on the command line when you use a command file:

```
lnk430 -r link.cmd c.obj d.obj
```

The linker processes the command file as soon as it encounters it, 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:

```
lnk430 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 that appear in a command file are insignificant except as delimiters. This applies to the format of linker directives in a command file, also. The following example shows a sample command file that contains linker directives. (Linker directive formats are discussed in later sections.)

```
/*************************************************************/
/*        Sample Linker Command File with Directives        */
/*************************************************************/
a.obj b.obj c.obj                    /* Input filenames      */
-o prog.out -m prog.map              /* Options              */

MEMORY                               /* MEMORY directives    */
{
  RAM:   origin = 200h      length = 0100h
  ROM:   origin = 0F000h    length = 1000h
}

SECTIONS                             /* SECTION directives   */
{
  .text:    > ROM
  .data:    > ROM
  .bss:     > RAM
}
```

Example 8.2: Command File With Linker Directives

The following names are reserved as keywords for linker directives. Do not use them as symbol or section names in a command file.

align	GROUP	origin
ALIGN	l (lowercase L)	ORIGIN
attr	len	page
ATTR	length	PAGE
block	LENGTH	range
BLOCK	load	run
COPY	LOAD	RUN
DSECT	MEMORY	SECTIONS
f	NOLOAD	spare
FILL	o	type
fill	org	TYPE
group		UNION

Constants in Command Files

Constants can be specified with either of two syntax schemes: the scheme used for specifying decimal, octal, or hexadecimal constants used in the assember or the scheme used for integer constants in "C" syntax.

8.5 Object Libraries

An object library is a partitioned archive file that contains complete 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 MSP430 archiver to build and maintain libraries.

Using object libraries can reduce link time and can reduce the size of the executable module. If a normal object file that contains a function is specified at link time, it is linked whether it is used or not; however, if that same function is placed in an archive library, it is included only if it is referenced.

The order in which libraries are specified is important because the linker includes only those members that resolve symbols that are undefined when the library is searched. The same library can be specified as often as necessary; it is searched each time it is included, or the -x option may be used. 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.

This example links several files and libraries. Assume the following:

- 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.
- Library libc.lib, member 0, contains a definition of origin.
- Library liba.lib, member 3, contains a definition of fillclr.
- Member 1 of both libraries defines clrscr.

If you enter lnk430 f1.obj liba.lib f2.obj libc.lib:

- Member 1 of liba.lib satisfies both references to clrscr because the library is searched and clrscr is defined before f2.obj references it.
- Member 0 of libc.lib satisfies the reference to origin.
- Member 3 of liba.lib satisfies the reference to fillclr.

If, however, you enter lnk430 f1.obj f2.obj libc.lib liba.lib, 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 -u option to force the linker to include a library member. The next example creates an undefined symbol rout1 in the linker's global symbol table:

```
lnk430 -u rout1 libc.lib
```

If any members of libc.lib define rout1, the linker includes those members. Note that it is not possible to control the allocation of individual library members; members are allocated according to the SECTIONS directive default allocation algorithm.

8.6 The MEMORY Directive

The linker determines where output sections should be allocated into memory; the linker must have a model of target memory to accomplish this task. 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 the model 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 the MEMORY directive to define a memory model, you can use the SECTIONS directive to allocate output sections into defined memory.

8.6.1 Default Memory Model

The linker's default memory model is based on the MSP430 architecture. This model assumes that the following memory is available:

- 256 bytes of RAM, beginning at location 200h
- 4K bytes of ROM, beginning at location 0F000h.

If you do not use the MEMORY directive, the linker uses this default memory model.

8.6.2 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 of memory 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 the memory ranges that are available to load code into. Any memory that you do not explicitly account for with the MEMORY directive 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. For example, you could use the MEMORY directive to specify a memory configuration as follows:

```
/**********************************************************/
/* Sample command file with MEMORY directive             */
/**********************************************************/
file1.obj     file2.obj    /* Input files */
-o prog.out             /* Options   */

MEMORY
{
     RAM:       origin =  200h      length =   100h
     ROM:       origin =  0F000h    length =  1000h
}
```

You could then use the SECTIONS directive to link the .bss section into the memory area named RAM, .text into ROM, and .data into ROM.

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 may be 1 to 8 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.

attr Specifies 1 to 4 attributes associated with the named range. Attributes are optional; when used, they must be enclosed in parentheses. Attributes can 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 include:

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 and may be abbreviated as *org* or *o*. The value, specified in bytes, is a long integer constant and may be decimal, octal, or hexadecimal.

length Specifies the length of a memory range and may be abbreviated as *len* or *l*. The value, specified in bytes, is a long integer constant and may be decimal, octal, or hexadecimal.

fill Specifies a fill character for the memory range and may be abbreviated as *f*. Fills are optional. The value is a two–byte integer constant and may be decimal, octal, or hexadecimal. The fill value will be used to fill any areas of the memory range that are not allocated to a section.

Note: Filling Memory Ranges

If you specify fill values for large memory ranges, your output file will be very large because filling a memory range (even with zeros) 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 = 02h, l = 0FEh, f = 0FFFFh
}
```

You normally use the MEMORY directive in conjunction with the SECTIONS directive to control allocation of output sections. After you use the MEMORY directive to specify the target system's memory model, you can use the SECTIONS directive to allocate output sections into specific named memory ranges or into memory that has specific attributes.

8.7 The SECTIONS Directive

The SECTIONS directive tells the linker how to combine sections from input files into sections in the output module and where to place the output sections in memory. In summary, the SECTIONS directive:

- 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.

8.7.1 Default Sections Configuration

If you do not specify a SECTIONS directive, the linker uses a default algorithm for combining and allocating the sections.

8.7.2 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.) After the section *name* is a list of properties that define the section's contents and how it is allocated. The properties may be separated by optional commas. Possible properties for a section are:

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 composing the section.

Syntax: { input_sections }

section type defines flags for special section types.

Syntax: **type = COPY** **or**
type = DSECT **or**
type = NOLOAD

For more information on section types, see Section 8.12.

fill value defines the value used to fill uninitialized "holes"

Syntax: **fill =** *value* **or**
name: ... { ... } = *value*

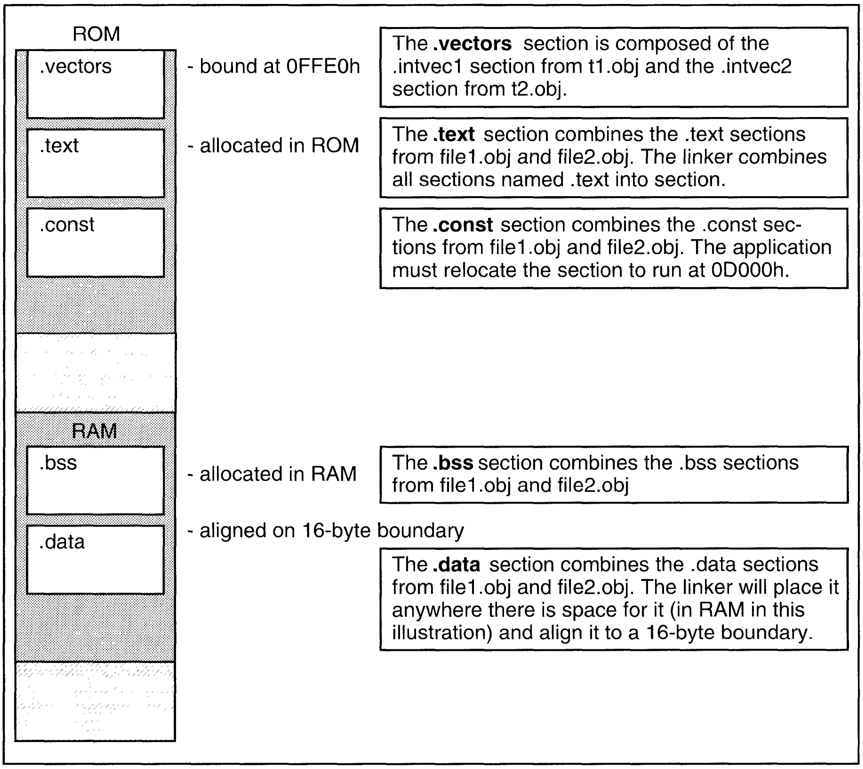
For more information on creating and filling holes, see Section 8.14.

The example shows a SECTIONS directive in a sample linker command file. The figure on the next page shows how these sections are allocated in memory.

```
/*************************************************************/
/* Sample command file with SECTIONS directives              */
/*************************************************************/
file1.obj file2.obj                     /* Input files       */
-o prog.out                             /* Options           */

SECTIONS
{
  .text:     load = ROM
  .const:    load = ROM, run = 0D000h
  .bss:      load = RAM
  .vectors: load = 0FFE0h
    {
      t1.obj(.intvec1)
      t2.obj(.intvec2)
      endvec = .;
    }
  .data:      align = 16
}
```

Example 8.3: The SECTIONS Directive

The figure shows the five output sections defined by the sections directive in the last example: .vectors, .text, .const, .bss, and .data.



The **.vectors** section is composed of the .intvec1 section from t1.obj and the .intvec2 section from t2.obj.

The **.text** section combines the .text sections from file1.obj and file2.obj. The linker combines all sections named .text into section.

The **.const** section combines the .const sections from file1.obj and file2.obj. The application must relocate the section to run at 0D000h.

The **.bss** section combines the .bss sections from file1.obj and file2.obj

The **.data** section combines the .data sections from file1.obj and file2.obj. The linker will place it anywhere there is space for it (in RAM in this illustration) and align it to a 16-byte boundary.

Figure 8.2: Section Allocation

8.7.3 Specifying the Address of Output Sections (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. In any case, the process of locating the output section in the target's memory and assigning its address(es) is called allocation.

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 equals sign or greater–than sign, and a value optionally enclosed in parentheses. If load and run allocation is separate, all parameters following the keyword LOAD apply to load allocation, and those following RUN apply to run allocation. Possible allocation parameters are:

binding	allocates a section at a specific address `.text: load = 0x1000`
memory	allocates the section into a range defined in the MEMORY directive with the specified name or attributes `.text: load > ROM`
alignment	specifies that the section should start on an address boundary `.text: align = 0x100`
blocking	specifies that the section must fit between two address boundaries: for example, on a single data page. `.text: block(0x100)`

For the load (usually the only) allocation, you may simply use a greater–than sign and omit the LOAD keyword:

```
.text: > ROM             .text: {...} > ROM
.text: > 0x4000
```

If more than one parameter is used, you can string them together as follows:

```
.text: > ROM align 16
```

Or if you prefer, use parentheses for readability:

```
.text: load = (ROM align(16))
```

Binding

You can supply a specific starting address for an output section by following the section name with an address:

```
.text: 0x4000
```

This example specifies that the .text section must begin at location 4000h. 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.

Note: Binding and Alignment or Named Memory Are Incompatible

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.

Memory

You can allocate a section into a memory range that is defined by the MEMORY directive. This example names ranges and links sections into them:

```
MEMORY
{
     ROM (RIX) :      origin = 0h,      length = 1000h
     RAM (RWIX):      origin = 0D000h,  length = 1000h
}
SECTIONS
{
     .text   :    > ROM
     .data   :    > RAM,         ALIGN=64
     .bss    :    > RAM
}
```

In this example, the linker places .text into the area called ROM. The .data and .bss output sections are allocated into RAM. You can align a section within a named memory range; the .data section is aligned on a 64–word boundary within the RAM 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 ROM or RAM area because both areas have the X attribute. The .data section can also go into either ROM or RAM because both areas have the R and I attributes. The .bss output section, however, must go into the RAM area because only RAM 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 other sections had been bound to addresses that would interfere with this allocation process, the .text section would start at address 0. If a section must start on a specific address, use binding instead of named memory.

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. For example:

```
.text: load = align(32)
```

allocates .text so that it falls on a 32-byte boundary.

Blocking is a weaker form of alignment that places a section so that it is allocated anywhere **within** a "block" of size *n*. As with alignment, *n* must be a power of 2. For example:

```
bss: load = block(0x1000)
```

allocates .bss so that the entire section is contained in a single 4K–byte data page.

You can use alignment or blocking alone or in conjunction with a memory area, but alignment and blocking cannot be used together.

8.7.4 Specifying Input Sections

An input section specification identifies the sections from input files that are combined to form an output section. The linker combines input sections by concatenating them in the order specified. The size of an output section is the sum of the sizes of the input sections that make up the output section.

```
SECTIONS
{
    .text:
    .data:
    .bss :
}
```

Example 8.4: The Most Common Method of Specifying Section Contents

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      */
     }
}
```

Note that it is not necessary for an input section to have the same name as another it is combined with or as the output section it becomes 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 the example on the page before are actually a shorthand method for the following:

```
SECTIONS
{
     .text: { *(.text) }
     .data: { *(.data) }
     .bss:  { *(.bss)  }
}
```

The *(.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 certain name, but the output section name is different from the input sections' names.
- You want the linker to allocate the input sections *before* it processes additional input sections or commands within the braces.

Here's an example that uses this method:

```
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*. Note that 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.

8.8 Specifying a Section's Runtime Address

It may be necessary or desirable at times to load code into one area of memory and run it in another. For example, you may have performance–critical code in a ROM–based system. The code must be loaded into ROM but would run much faster if it were in RAM.

The linker provides a simple way to specify this. In the SECTIONS directive, you can optionally 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 = ROM, run = RAM
```

Use the *load* keyword for the load address and the *run* keyword for the run address.

8.8.1 Specifying Two Addresses

The load address determines where a loader will place 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 just by specifying a separate run address.

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

If either the load or run address has additional parameters, such as alignment or blocking, list them after the appropriate keyword. After the keyword *load,* everything having to do with allocation 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 may also specify run first, then load. Use parentheses to improve readability. Examples:

```
.data: load = ROM, align = 32, run = RAM
```

(align applies only to load)

```
.data: load = (ROM align 32), run = RAM
```

(identical to previous example)

```
.data: run     =    RAM, align 32,
       load    =    align 16
```

(align 32 in RAM for run; align 16 anywhere for load)

8.8.2 Uninitialized Sections

Uninitialized sections (such as .bss) are not loaded, so the only address of significance 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. Examples:

```
.bss: load = 0x1000, run = RAM
```

A warning is issued, load is ignored, space is allocated in RAM. All of the following examples have the same effect. The .bss section is allocated in RAM.

```
.bss: load = RAM
.bss: run = RAM
.bss: > RAM
```

8.8.3 Referring to the Load Address by Using the .label Directive

Any reference to a normal symbol in a section refers to its runtime address. However, it may be necessary at runtime to refer to a load–time address. In particular, the code that copies a section from its load address to its run address must know where it was loaded. The .label directive in the assembler defines a special type of symbol that refers to the load address of the section. Thus, whereas normal symbols are relocated with respect to the run address, .label symbols are relocated with respect to the load address.

```
;------------------------------------------------------------------
;   define a section to be copied from ROM to RAM
;------------------------------------------------------------------
                .sect ".fir"
                .label fir_load       ; load address of section
fir:                                  ; run address of section
;               <code here>           ; code for the section
                ret

                .label fir_end

fir_len .equ fir_end - fir_load

;------------------------------------------------------------------
;  copy .fir section from ROM into RAM
;------------------------------------------------------------------
                .text
                MOV   #fir_len, R4
                MOV   #fir_load, R5
                MOV   #fir, R6

                JMP   L2
L1:             MOV   @R5+, 0(R6)
                INCD  R6
L2:             DECD  R4
                JC    L1


;------------------------------------------------------------------
;  jump to section, now in RAM
;------------------------------------------------------------------
                call fir              ; call runtime address
```

Linker Command File

```
/***********************************************************/
/* PARTIAL LINKER COMMAND FILE FOR FIR EXAMPLE             */
/***********************************************************/
MEMORY
{
  ROM:    origin=4000h     length = 4000h
  RAM:    origin=2000h     length = 2000h
}

SECTIONS
{
  .text:  load = ROM
  .fir:   load = ROM, run = RAM
}
```

Example 8.5: Copying a Section From ROM to RAM

The figure illustrates the runtime execution of the last example.

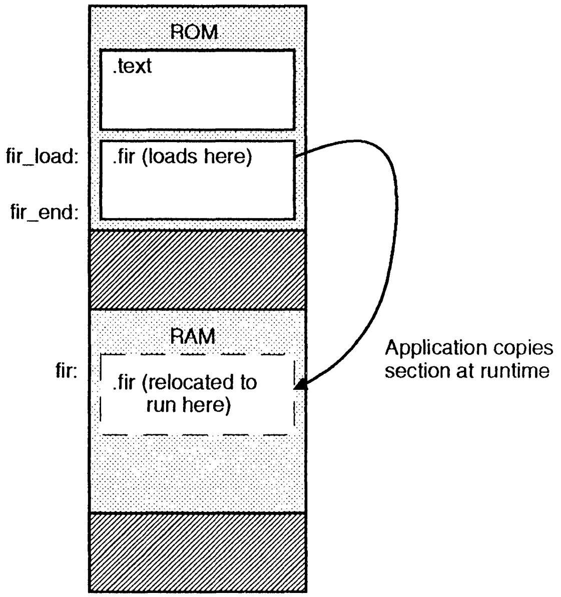

ROM
.text
fir_load:
.fir (loads here)
fir_end:
RAM
fir:
.fir (relocated to run here)
Application copies section at runtime

Figure 8.3: Runtime Execution

8.9 Using UNION and GROUP Statements

Two SECTIONS statements allow you to conserve memory: GROUP and UNION. Unioning output sections causes the linker to allocate the same run address to the sections. Grouping output sections causes the linker to allocate them contiguously in memory.

8.9.1 Overlaying Sections With the UNION Statement

For some applications, you may wish to allocate more than one section to run at the same address; for example, you may have several routines you want in on–chip RAM at various stages of the program's execution. Or you may want several data objects that you know will not be 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 address.

```
SECTIONS
{
   .text: load = ROM
   UNION: run  = RAM
   {
               .bss1: { file1.obj(.bss) }
               .bss2: { file2.obj(.bss) }
   }
   .bss3: run = RAM { globals.obj(.bss) }
}
```

Example 8.6: Illustration of the Form of the UNION Statement

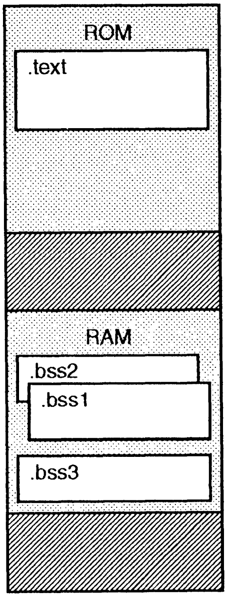


Figure 8.4: Runtime Memory Allocation

In the example on the page before, the .bss sections from file1.obj and file2.obj are allocated *at the same address* in RAM. The union occupies as much space in the memory map as its largest component. The components of a union remain independent sections; they are simply allocated together as a unit.

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 has raw data, such as .text), its load allocation **must** be separately specified. The next example illustrates this.

```
UNION run = RAM
{
   .text1: load = ROM, { file1.obj(.text) }
   .text2: load = ROM, { file2.obj(.text) }
}
```

Example 8.7: Illustration of Separate Load Addresses for UNION Sections

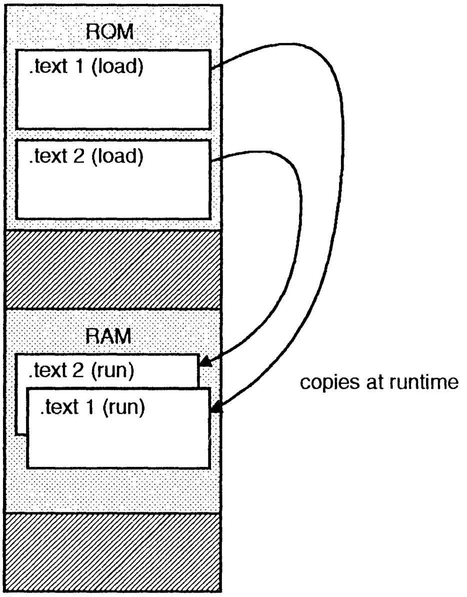


Figure 8.5: Load and Run Memory Allocation

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 fits 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 redundant 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 are specified, the linker issues a warning and ignores the load address.

8.9.2 Grouping Output Sections Together

The SECTIONS directive has a GROUP option that 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:

```
SECTIONS
{
    .text                /* Normal output section               */
    .bss                 /* Normal output section               */
    GROUP 1000h :        /* 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 way a single output section is allocated. In the preceding example, the GROUP is bound to address 1000h. This means that .data is allocated at 1000h, and *term_rec* follows it in memory.

Note: You Cannot Specify Addresses for Sections Within a Group

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.

8.11 Default Allocation Algorithm

The MEMORY and SECTIONS directives provide flexible methods for building, combining, and allocating sections. Any memory locations or sections that you choose *not* to specify, however, must still be handled by the linker. The linker uses default algorithms to build and allocate sections within the specifications you supply.

8.11.1 Default Allocation

If you do not use any MEMORY or SECTIONS directives, the linker acts as though the following definitions were specified:

```
MEMORY
{
     RAM    :     origin = 200h   length = 100h
     ROM    :     origin = 0F000h length = 1000h
}
SECTIONS
{
     .bss   :     >    RAM
     .text  :     >    ROM
     .data  :     >    ROM

}
```

All .bss input sections are concatenated to form one .bss output section linked into. All .data input sections are combined to form a .data output section, which is linked into ROM. All .text input sections are concatenated to form a .text output section, which is linked into ROM starting at location 0F000h.

Unless you specify otherwise with a MEMORY directive, the linker assumes the configuration specified above. That is, the only memory that the linker uses to build your program is:

- 256 bytes starting at location 0200h,
- 4K bytes starting at location 0F000h.

If there are additional input sections in the input files (specifically, named sections), the linker links them in after the default sections have been linked. Input sections that have the same name are combined into a single output section with this name. The linker allocates these additional output sections into memory wherever there is room. Usually, it is desirable to use explicit SECTIONS directives to tell the linker where to place named sections.

Note: The SECTIONS Directive

If a SECTIONS directive is specified, 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 below.

8.11.2 General Rules for Forming Output Sections

An output section can be formed in one of two ways:

Rule 1 As the result of a SECTIONS directive definition.

Rule 2 By combining input sections with the same names into output sections that are not defined in a SECTIONS directive.

If an output section is formed as a result of a SECTIONS directive (rule 1), this definition completely determines the section's contents.

An output section can also be formed when input sections are encountered that are not specified by any SECTIONS directive (rule 2). In this case, the linker combines all such input sections that have the same name into an output section with this 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 to contain 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.

After the linker determines the composition of all the output sections, it must allocate them into configured memory. The MEMORY directive specifies which portions of memory are configured, or if there is no MEMORY directive, the linker uses the default configuration.

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. This is the algorithm:

1) Output sections for which you have supplied a specific binding address are placed in memory at that address.
2) Output sections that are included in a specific, named memory range or that have memory attribute restrictions are allocated. Each output section is placed into the first available space within the named area, considering alignment where necessary.
3) Output sections that have zero length are allocated at the beginning of the first appropriate memory area unless they are part of a group.
4) 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.

8.12 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. For example:

```
SECTIONS
{
    sec1: load = 2000h, type =  DSECT     {f1.obj}
    sec2: load = 4000h, type =  COPY      {f2.obj}
    sec3: load = 6000h, type =  NOLOAD    {f3.obj}
}
```

- The DSECT type creates a “dummy section" that has the following qualities:
  - 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 2000h. 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.
- 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 it, it appears in the memory map listing, etc.

8.13 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.

8.13.1 Syntax of Assignment Statements

The syntax of assignment statements in the linker is similar to that of assignment statements in the C language:

symbol	=	*expression;*	assigns the value of expression to symbol
symbol	+ =	*expression;*	adds the value of expression to symbol
symbol	- =	*expression;*	subtracts the value of expression from symbol
symbol	* =	*expression;*	multiplies symbol by expression
symbol	/ =	*expression;*	divides symbol by expression

The symbol should be defined externally in the program. If it is not, the linker defines a new symbol and enters it into the symbol table. Assignment statements **must** be terminated 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. cur_tab must point to either Table1 or Table2. You could accomplish this in the assembly code, but you would need to reassemble the program in order 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 */
```

8.13.2 Assigning the SPC to a Symbol

A special symbol, denoted by a dot (.), represents the current value of the SPC during allocation. 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.

For example, suppose a program needs to know the address of the beginning of the .data section. By using the .global directive, 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 ultimate linked address of the .data section. (dstart is assigned *before* .data is allocated.) The linker will relocate all references to Dstart.

A special type of assignment assigns a value to the "." symbol. This adjusts the location counter 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 ".".

8.13.3 Assignment Expressions

These rules apply to linker expressions:

- Expressions can contain global symbols, constants, and the C language operators listed in the next table.
- All numbers are treated as long (32–bit) integers.
- Constants are identified by the linker in the same manner as they are 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 zero 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, the symbol is relocatable; if it is assigned the value of an absolute expression, the symbol is absolute.

The linker supports the C language operators listed in the table in order of precedence. Operators in the same group have the same precedence. Besides the operators listed in the table, 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 expression

```
. = align(16);
```

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.

Group 1 (Highest Precedence)		**Group 6**	
!	Logical not	&	Bitwise AND
~	Bitwise not		
-	Negative		
Group 2		**Group 7**	
*	Multiplication	|	Bitwise OR
/	Division		
%	Mod		
Group 3		**Group 8**	
+	Addition	&&	Logical AND
-	Minus		
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		

Table 8.2: Operators in Assignment Expressions

8.13.4 Symbols Defined by the Linker

The linker automatically defines several symbols that a program can use at runtime to determine where a section is linked. Since these symbols are external, they appear in the link map. Each symbol can be accessed in any assembly language module if it is declared with a .global directive. Values are assigned to these symbols as follows:

.text	is assigned the first address of the .text output section. (It marks the *beginning* of executable code.)
etext	is assigned the first address following the .text output section. (It marks the *end* of executable code.)
.data	is assigned the first address of the .data output section. (It marks the *beginning* of initialized data tables.)
edata	is assigned the first address following the .data output section. (It marks the *end* of initialized data tables.)
.bss	is assigned the first address of the .bss output section. (It marks the *beginning* of uninitialized data.)

end is assigned the first address following the .bss output section. (It marks the *end* of uninitialized data.)

8.14 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 such holes and how you can fill holes (and uninitialized sections) with values.

8.14.1 Initialized and Uninitialized Sections

There are two guidelines to remember about the contents of an output section. An output section contains either:

- Raw data for the *entire* section, **or**
- *No* raw data.

A section that has raw data is said to be **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 and sections defined with the .usect directive 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 RAM for variables. In the object file, an uninitialized section has a normal section header and may have symbols defined in it; no memory image, however, is stored in the section.

8.14.2 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 follow the first guideline (above) and supply raw data for the hole.*

Holes can be created only *within* output sections. There can also be space *between* output sections, but such spaces are not holes.

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 following example uses assignment statements to create holes in output sections:

```
SECTIONS
{
     outsect:
     {
          file1.obj(.text)
          . += 100h;       /* Create a hole with size 100h    */
          file2.obj(.text)
          . = align(16);   /* Create a hole to align the SPC */
          file3.obj(.text)
     }
}
```

The output section outsect is built as follows:

- The .text section from file1.obj is linked in.
- The linker creates a 256–byte hole.
- The .text section from file2.obj is linked in after the hole.
- The linker creates another hole by aligning the SPC on a 16–byte boundary.
- 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 file3.obj .text to start on a 16–word boundary within outsect. If outsect is ultimately allocated to start on an address that is not aligned, file3 .text will not be aligned, either.

Expressions that decrement " . " are illegal. For example, it is invalid to use the -= operator in an assignment to " . ". The most common operators used in assignments to " . " 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. For example:

```
.text:    {     .+= 100h; }       /* Hole at the beginning    */
.data:    {
                *(.data)
                . += 100h;   }    /* 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.* Here is an example of creating a hole in this way:

```
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 (first guideline). Therefore, the uninitialized .bss section becomes a hole.

Note that 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.

8.14.3 Filling Holes

Whenever there is a hole in an initialized output section, the linker must supply raw data to fill it. The linker fills holes with a 16–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 16–bit constant. For example:

```
SECTIONS
{
          outsect:
     {
          file1.obj(.text)
          file2.obj(.bss) = 0FFh /* Fill this hole */
     }                           /* with 00FFh     */
}
```

2) You can also specify a fill value for all the holes in an output section by using the fill keyword. For example:

```
SECTIONS
{
     outsect: fill = 0FF00h      /* This fills holes       */
                                 /* with 0FF00h               */
     {
          . += 10h;              /* 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 -f. Suppose the command file link.cmd contains the following SECTIONS directive. For example:

   ```
   SECTIONS
   {
         .text: { .= 100; }    /* Create a 100-word hole */
   }
   ```

   Now invoke the linker with the -f option:

   ```
   lnk430 -f 0FFFFh link.cmd
   ```

   This fills the hole with 0FFFFh.

4) If you do not invoke the linker with the -f option, 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.

8.14.4 Explicit Initialization of Uninitialized Sections

An uninitialized section becomes a hole only when it is combined with an initialized section. When uninitialized sections are combined with each other, the resulting output section remains uninitialized and has no raw data in the output file.

However, 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 = 1234h         /* Fills .bss with 1234h */
}
```

Note: Filling Sections

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.

8.14.5 Examples of Using Initialized Holes

The MSP430X201 device has 4K bytes of program memory, starting at location 0F000h. The top bytes of this area are reserved for interrupt vectors. Suppose you want to link the .text sections from three object files into a .text output section that begins at address 0F000h. Suppose also that you have a section of initialized interrupt vectors called int_vecs that you want to link at address 0FFE0h. You could fill the space between the end of the .text section and the beginning of the interrupt vectors; the figure shows the space filled with a 1–byte fill value of 0EFh and illustrates the desired memory map for program memory.

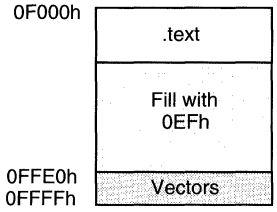


Figure 8.6: Initialized Hole

To obtain the configuration shown in the figure, you must create one large output section that has .text at the beginning, int_vecs at the end, and a hole between filled with 0EFh:

```
SECTIONS
{
    prog 0F000h :fill = 0EFEFh     /* Define prog and start at 0F000h and */
         {                         /* Specify a fill value                */
         file1.obj(.text)          /* Link .text sections from each file  */
         file2.obj(.text)
         file3.obj(.text)
         . = 0FE0h;                /* Create hole to 0FE0h (0FFE0h abs)   */
         file1.obj(int_vecs)       /* Link in the vectors section         */
    }
}
```

The fill value must be a 16–bit constant. To have the value 0EFh in each byte, the fill value was specified as 0EFEFh.

Notice that the value 0FE0h, which is assigned to the section program counter (.), is relative to the beginning of the section. Because the section begins at 0F000h, the hole is actually created from the end of the .text section to address 0FFE0h.

8.15 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:

- Intermediate files **must** have relocation information. Use the -r option when you link the file the first time.
- Intermediate files **must** have symbolic information. By default, the linker retains symbolic information in its output. Do not use the -s option if you plan to relink a file, because -s strips symbolic information from the output module.
- Intermediate link steps 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 link step.

The following example shows how you can use partial linking:

Step 1: Link the file file1.com; use the -r option to retain relocation information in the output file tempout1.out.

```
lnk430 -r -o tempout1 file1.com
```

file1.com contains:

```
SECTIONS
{
     ss1:    {
             f1.obj
             f2.obj
               .
               .
               .
             fn.obj
             }
}
```

Step 2: Link the file file2.com; use the -r option to retain relocation information in the output file tempout2.out.

```
lnk430 -r -o tempout2 file2.com
```

file2.com contains:

```
SECTIONS
{
     ss2:    {
             g1.obj
             g2.obj
               .
               .
               .
             gn.obj
             }
}
```

Step 3: Link tempout1.out and tempout2.out:

```
lnk430 -m final.map -o final.out tempout1.out tempout2.out
```

8.17 Linker Example

This example links a program called demo.out. There are three object modules, demo.obj, ctrl.obj, and tables.obj.

Assume the following memory configuration:

Address Range	**Memory Contents**
200h to 2FFh	internal RAM
1F00h to 1FFFh	Data EEPROM
2000h to 3FF**Fh**	8K external RAM
0F000h to 0FFFFh	4K internal program ROM

The program is built from the following elements:

- Executable code, contained in the .text sections of demo.obj and ctrl.obj, must be linked into program ROM. The symbol SETUP must be defined as the program entry point.
- A set of interrupt vectors, contained in the int_vecs section of tables.obj, must be linked at address 0FFE0h in program ROM.
- A table of coefficients, contained in the .data section of tables.obj, must be linked into .EEPROM. The remainder of EEPROM must be initialized with the value 0A26Eh.
- A set of variables, contained in the .bss section of ctrl.obj, must be linked into the RAM. These variables must be preinitialized to 0FFFFh.
- Another .bss section in demo.obj must be linked into external RAM.

The next two figures illustrate the linker command file and the map file for this example.

```
/*******************************************************************/
/* Specify the Linker Options                                      */
/*******************************************************************/
-e SETUP               /* Define the entry point                   */
-o demo.out            /* Name the output file                     */
-m demo.map            /* Create a load map                        */
/*******************************************************************/
/* Specify the Input Files                                         */
/*******************************************************************/
demo.obj
ctrl.obj
tables.obj
/*******************************************************************/
/* Specify the Memory Configuration                                */
/*******************************************************************/
MEMORY
{
    RAM        :  origin    =     0200h    length = 0100h
    EEPROM     :  origin    =     1F00h    length = 0100h
    RAMEXT     :  origin    =     2000h    length = 2000h
    ROM        :  origin    =    0F000h    length = 1000h
}
/*******************************************************************/
/* Specify the Output Sections                                     */
SECTIONS
    .text: > ROM            /* Link all .text sections into ROM     */

    int_vecs 0FFE0h: {}     /* Link interrupts at FFE0h             */

    .data:                  /* Link the data sections               */
    {
        tables.obj(.data)
        . = 100h;           /* Create a hole to end of the block    */
    } = 0A26Eh > EEPROM     /* Fill and link into EEPROM            */

    ctrl_vars:              /* Create new section for ctrl vars     */
    {
        ctrl.obj(.bss)
    } = 0FFFFh > RAM        /* Fill with 0FFFFh and link to RAM     */

    .bss    > RAMEXT        /* Link all remaining .bss sections     */
}
/*******************************************************************/
/* End of Linker Command File                                      */
/*******************************************************************/
```

Figure 8.7: Linker Command File, demo.cmd

Now invoke the linker by entering the following command:

lnk430 demo.cmd

This creates the map file shown in the next figure and an output file called demo.out that can be run on the MSP430.

```
*****************************************************************
MSP430 COFF Linker                                   Version 1.00
*****************************************************************
Thu Feb 10 09:21:32 1994
OUTPUT FILE NAME:   <demo.out>
ENTRY POINT SYMBOL: "SETUP"  address: 0000f000

MEMORY CONFIGURATION
     name        origin     length      attributes     fill
     RAM         00000200   000000100    RWIX
     EEPROM      00001f00   000000100    RWIX
     RAMEXT      00002000   000002000    RWIX
     ROM         0000f000   000001000    RWIX

SECTION ALLOCATION MAP

output                                  attributes/
section  page   origin     length       input sections
.text       0 0000f000   00000010
              0000f000   00000008       demo.obj (.text)
              0000f008   00000000       tables.obj (.text)
              0000f008   00000008       ctrl.obj (.text)

int_vecs    0 0000ffe0   00000020
              0000ffe0   00000020       tables.obj (int_vecs)

.data       0 00001f00   00000100
              00001f00   00000008       tables.obj (.data)
              00001f08   000000f8       --HOLE-- [fill = a26e]
              00002000   00000000       ctrl.obj (.data)
              00002000   00000000       demo.obj (.data)

ctrl_var    0 00000200   00000004
              00000200   00000004       ctrl.obj (.bss) [fill = ffff]

.bss        0 00002000   00000004       UNINITIALIZED
              00002000   00000004       demo.obj (.bss)
              00002004   00000000       tables.obj (.bss)

GLOBAL SYMBOLS
address    name                         address     name
00002000   .bss                         00001f00     .data
00001f00   .data                        00002000      edata
0000f000   .text                        00002000      .bss
0000f000   SETUP                        00002004      end
00002000   edata                        0000f000     .text
00002004   end                          0000f000     SETUP
0000f010   etext                        0000f010      etext

[7 symbols]
```

Figure 8.8: Output Map File, demo.map

Topics

Figures

9 Absolute Lister Description

The MSP430 absolute lister is a debugging tool. This utility 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. Normally, this is a tedious process requiring many manual operations; the absolute lister utility, however, performs these operations automatically.

9.1 Producing an Absolute Listing

The figure illustrates the steps required to produce an absolute listing.

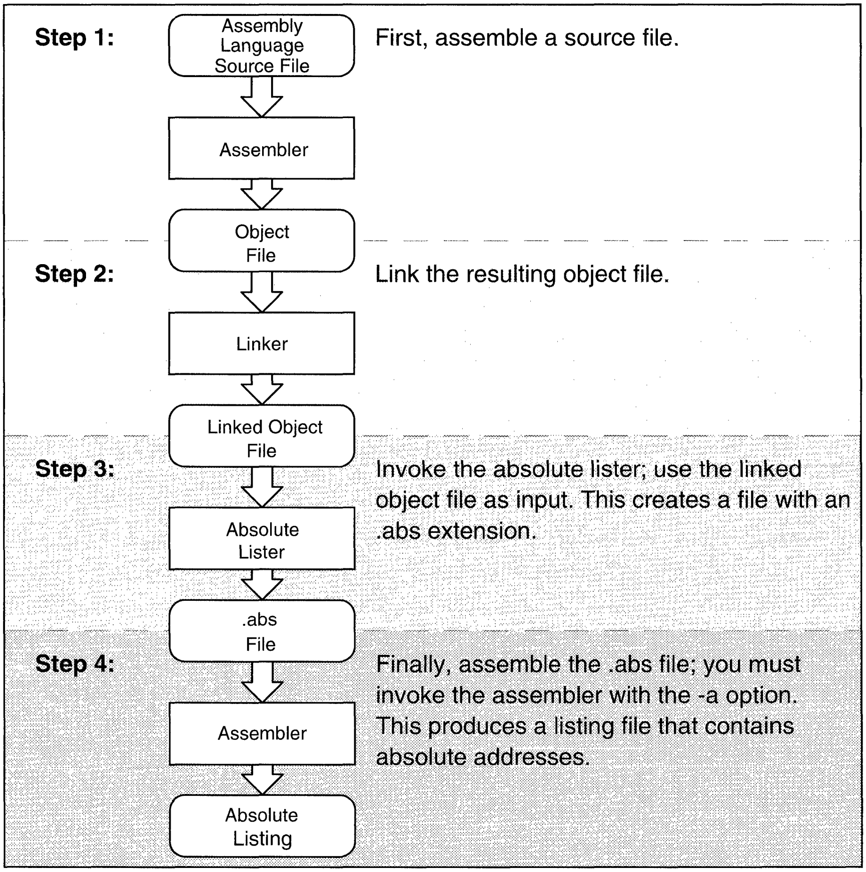

Step 1:
Assembly Language Source File
Assembler
Object File
First, assemble a source file.
Step 2:
Link the resulting object file.
Linker
Linked Object File
Step 3:
Invoke the absolute lister; use the linked object file as input. This creates a file with an .abs extension.
Absolute Lister
.abs File
Step 4:
Finally, assemble the .abs file; you must invoke the assembler with the -a option. This produces a listing file that contains absolute addresses.
Assembler
Absolute Listing

Figure 9.1: Absolute Lister Development Flow

9.2 Invoking the Absolute Lister

The syntax for invoking the absolute lister is:

abs430 *filename*

where *filename* must be a linked object file. The absolute lister assumes that this file has an extension of .out. (This is the extension that the linker produces for output files).

If you omit the filename when you invoke the absolute lister, the utility prompts you for a filename.

The absolute lister produces an output file for each file that was linked to create *filename*.out. These files are named with the individual filenames and an extension of **.abs**.

Assemble this file and use the -a assembler option to create the absolute listing:

```
asm430 filename.abs -a
```

9.3 Absolute Lister Example

This example uses three source files. Note that module1.asm and module2.asm both include the file globals.def.

module1.asm	module2.asm	globals.def
.bss xflags,2	.copy "globals.def"	.global flags
.bss flags	.text	Gflag .set 2
.copy "globals.def"	bic #Gflag,&flags	
.text		
bis #Gflag,&flags		

The following steps create absolute listings for the files module1.asm and module2.asm:

Step 1: First, assemble module1.asm and module2.asm:

```
asm430 module1
asm430 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 abstest.cmd:

```
/***************************************************/
/* File abstest.cmd -- COFF linker control file     */
/* for linking MSP430 modules                       */
/***************************************************/
-o ABSTEST.OUT          /* executable output file    */
-m ABSTEST.MAP          /* output map file           */

/* input files                                      */
MODULE1.OBJ
MODULE2.OBJ

/* define MSP430 memory map                         */
MEMORY
{
       RAM:          origin=00200h  length=0100h
       ROM:          origin=0F000h  length=1000h
}

/* define the output sections                        */
SECTIONS
{
       .bss:            >RAM
       .text:           >ROM
}
```

Invoke the linker:

lnk430 abstest.cmd

This creates an executable object file called abstest.out; use this new file as input for the absolute lister.

Step 3: Now, invoke the absolute lister:

```
abs430 abstest.out
```

This creates two files called module1.abs and module2.abs:

module1.abs:

```
                        .nolist
flags                   .setsym     0202h
.bss                    .setsym     0200h
end                     .setsym     0203h
.text                   .setsym     0f000h
etext                   .setsym     0f008h
.data                   .setsym     00h
edata                   .setsym     00h
                        .setsect    ".text",0f000h
                        .setsect    ".data",00h
                        .setsect    ".bss",0200h
                        .list
                        .text
                        .copy       "MODULE1.ASM"
```

module2.abs:

```
                        .nolist
flags                   .setsym     0202h
.bss                    .setsym     0200h
end                     .setsym     0203h
.text                   .setsym     0f000h
etext                   .setsym     0f008h
.data                   .setsym     00h
edata                   .setsym     00h
                        .setsect    ".text",0f004h
                        .setsect    ".data",00h
                        .setsect    ".bss",0203h
                        .list
                        .text
                        .copy       "MODULE2.ASM"
```

These files have information that the assembler needs when you invoke it in step 4:

- They contain .setsym directives, which equate values to global symbols. Both files contain global equates for the symbol flags. The symbol flags 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 tell the assembler which assembly language source file to include.

Note that the .setsym and .setsect directives are not useful in normal assembly; they are useful only for creating absolute listings.

Step 4: Finally, assemble the .abs files created by the absolute lister (remember that you must use the -a option when you invoke the assembler):

```
asm430  -a  module1.abs
asm430  -a  module2.abs
```

This 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:

```
MSP430 Macro Assembler  Version 1.00 [04/94]     Wed May 25 14:12:55 1994
 Copyright (c) 1994    Texas Instruments Incorporated

MODULE1.ABS                                                   PAGE    1

       13 f000                             .text
       14                                  .copy        "MODULE1.ASM"
 A      1 0200                             .bss    xflags,2
 A      2 0202                             .bss    flags
 A      3                                  .copy   "globals.def"
 B      1                                  .global flags
 B      2         02               Gflag   .set    2
 B      3
 B      4
 A      4 f000                             .text
 A      5 f000   -d3a20202                 bis     #Gflag, &flags
 A      6

 No Errors,  No Warnings
```

module2.lst:

```
MSP430 Macro Assembler  Version 1.00 [04/94]     Wed May 25 14:42:20 1994
 Copyright (c) 1994    Texas Instruments Incorporated

MODULE2.ABS                                                          PAGE    1

       13 f004                          .text
       14                               .copy        "MODULE2.ASM"
 A      1                               .copy   "globals.def"
 B      1                               .global flags
 B      2        02             Gflag   .set    2
 B      3
 B      4
 A      2 f004                          .text
 A      3 f004  !c3a20202               bic     #Gflag, &flags
 A      4

 No Errors,  No Warnings
```

Topics

Figures

10 Object Format Converter Description

Most EPROM programmers do not accept COFF object files as input. The object format converter converts a COFF object file into one of four object formats that most EPROM programmers accept as input:

Extended Tektronix hex object format supports 32–bit addresses.

Intel hex object format supports 16–bit addresses.

Motorola S format supports 16–bit addresses.

TI–tagged object format supports 16–bit addresses.

10.1 Object Format Converter Development Flow

The figure illustrates the object format converter's role in the assembly language development process.

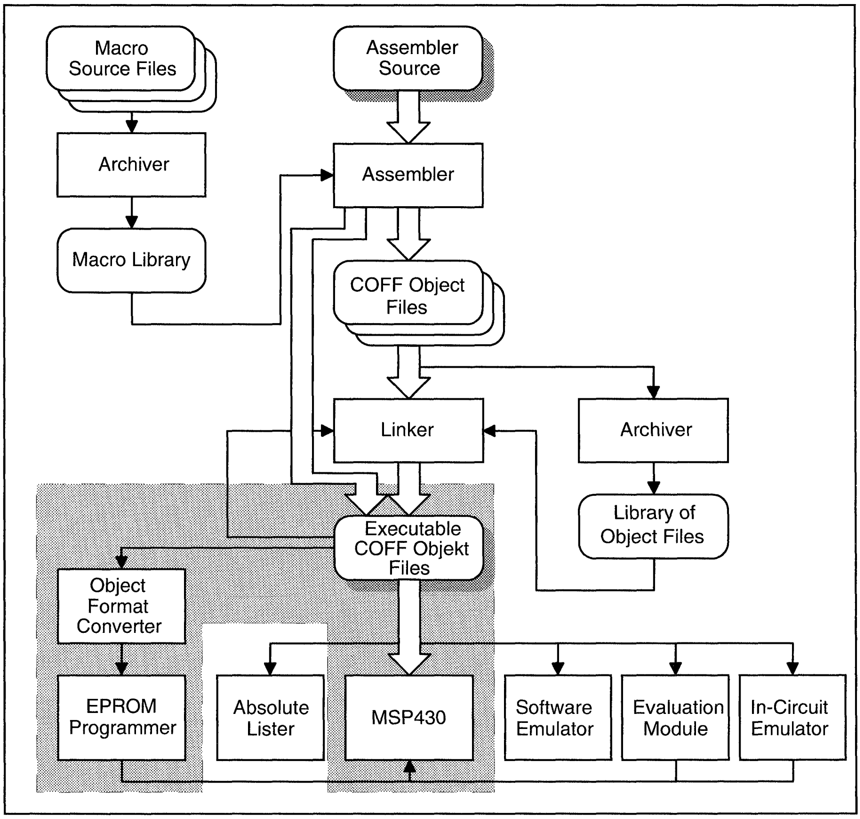


Figure 10.1: Object Format Converter Development Flow

10.2 Extended Tektronix Hex Object Format

The Extended Tektronix hex object format supports 32–bit addresses and has three types of records: data, symbol, and termination records.

termination record	signifies the end of a module.
symbol record	contains information about program sections.
data record	contains the header field, the load address, and the object code.

The header field, in the data record, contains the following information:

	Number of ASCII Characters	Description
%	1	Data type is Extended Tektronix hex format
Block length	2	Number of characters in the record, minus the %
Block type	1	6 = data record 8 = termination record
Sumcheck	2	A 2–digit hex sum modulo 256 of all values in the record except the % and the sumcheck itself.

The load address, in the data record, specifies where the object code will be located. The first number specifies the address length; this is always 8. The remaining characters of the data record contain the object code, 2 characters per byte.

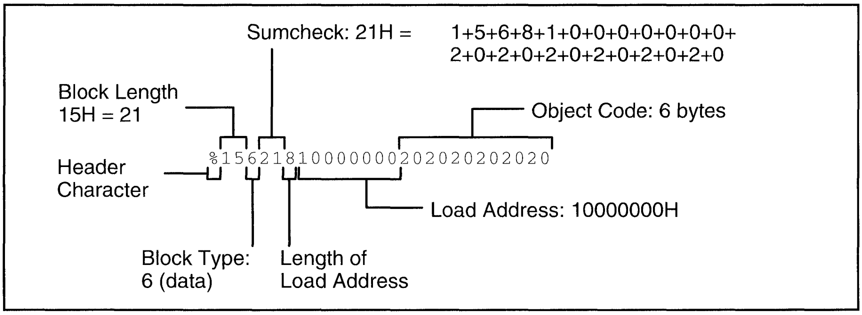


Figure 10.2: Extended Tektronix Hex Object Format

10.3 Intel Hex Object Format

The Intel hex object format supports 16–bit addresses and consists of a 9–character (4–field) prefix, which defines the start of record, byte count, load address, and record type, and a 2–character sumcheck suffix.

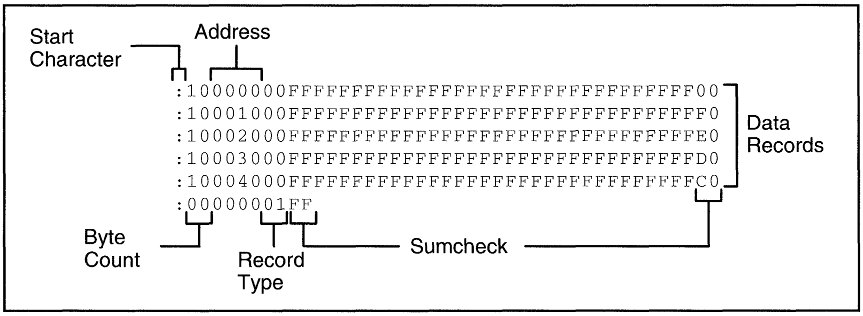
The two record types, which are represented in the 9–character prefix, are described below:

00 = data record (begins with the colon start character)

01 = end–of–file record

Record type *00*, the data record, begins with the colon (:) start character and is followed by the byte count, the address of the first data byte, the record type (00), and the sumcheck. The sumcheck is the 2s complement (in binary) 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 the colon (:) start character. The colon is followed by the byte count, the address, the record type (01), and the sumcheck.



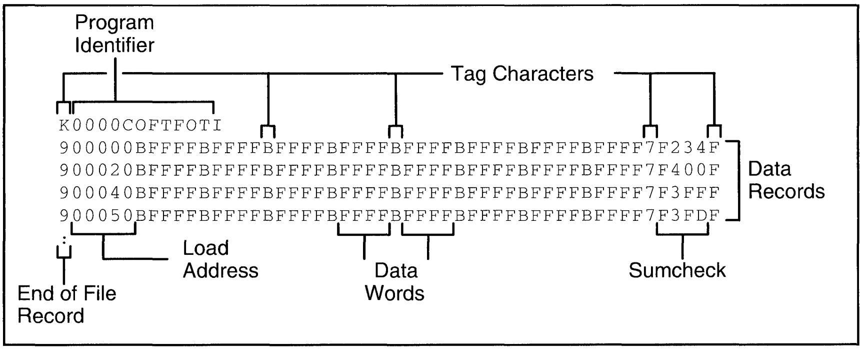
```
:10000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00
:10001000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0
:10002000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE0
:10003000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD0
:10004000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC0
:00000001FF
```

Figure 10.3: Intel Hex Object Format

10.4 TI–Tagged Object Format

The TI–tagged object format supports 16–bit addresses and consists of a start–of–file record, data records, and end–of–file record. Each of the data records is made up of a series of small fields and is signified by a tag character. The following is a list of the significant tag characters:

K	is followed by the program identifier.
7	is followed by a sumcheck (acknowledged).
8	is followed by a sumcheck (ignored).
9	is followed by a load address.
B	is followed by a data word (4 characters).
F	identifies the end of the data record.



```
K0000COFTFOTI
900000BFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFF7F234F
900020BFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFF7F400F
900040BFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFF7F3FFF
900050BFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFFBFFFF7F3FDF
:
```

Figure 10.4: TI–Tagged Object Format

If any data fields appear before the first address, the first field is assigned address 0000. Address fields may be expressed for any data byte, but none is required. The sumcheck field, which is preceded by the tag character **7**, is a 2s complement of the sum of the 8–bit ASCII values of characters, beginning with the first tag character and ending with the sumcheck tag character (7 or 8). The end–of–file record is a (:) colon.

10.5 Motorola S Format

The Motorola S format supports 16–bit addresses and consists of a start–of –file record, data records, and an end–of–file record. Each record is made up of five fields: record type, byte count, address, data, and sumcheck. The three record types are as follows:

- **S0** Header record
- **S1** Code/data record
- **S9** Termination record

The byte count is the character pair count in the record, excluding the type and byte count itself.

The sumcheck is the least significant byte of the ones complement of the sum of the values represented by the pairs of characters making up the byte count, address, and the code/data fields.

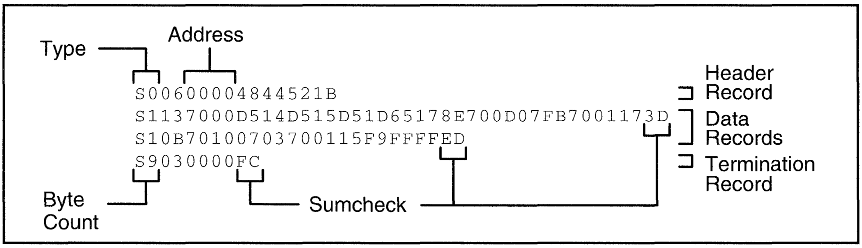

```
S00600004844521B
S1137000D514D515D51D65178E700D07FB7001173D
S10B70100703700115F9FFFFED
S9030000FC
```

Figure 10.5: Motorola S Format

10.6 Invoking the Object Format Converter

To invoke the object format converter, enter

```
rom430 [-option] [COFF input file [output file] ]
```

rom430 is the command that invokes the object format converter; all parameters are optional.

options can be entered anywhere on the line, but the order of filenames is significant. The filenames (if used) are interpreted as:

1) The input filename.

2) The output filename.

- These are the options:

  -x specifies Tektronix hex object format for the output.

  -i specifies Intel hex object format for the output.

  -t specifies TI–tagged object format for the output.

  -m specifies Motorola S format for the output.

  If you don't specify an option, the object format converter produces Tektronix hex format output files.

- If you do not specify an input filename, the object format converter prompts for it. If you specify a filename without an extension, the utility assumes that the filename has a default extension of ***.obj***.

- If you do not specify a second filename, the object format converter uses the input filename with an extension based on the format chosen:

Option	Format	Extension
-x	Tektronics	.tek
-i	Intel Hex	.int
-t	TI–Tagged	.tag
-m	Motorola S	.ms

When the utility finishes converting the input file, it prints the message *Translation complete*.

10.7 Object Format Converter Examples

Here are some examples of using the object format converter.

- **Example 1**

  You can invoke the object format converter with no options and no filenames by entering:

  ```
  rom430
  ```

  The utility will print the following banner and prompt:

  ```
  COFF Object Converter      Version 1.00
  Copyright (c) 1994, Texas Instruments Incorporated
        Coff file [.obj]:
  ```

  If, for example, you respond to the prompt with a filename of test, the object format converter uses the file test.obj as an input file. The utility produces an output file named test.tek in Tektronix hex format. (Tektronix format is the default when you don't specify a format.)

- **Example 2**

  If you enter

  ```
  rom430 -i in out1
  ```

  the utility uses in.obj as the input file. It creates an Intel hex format file named out1.in.

- **Example 3**

  If you enter

  ```
  rom430 -x in.tmp out.x
  ```

  the object format converter uses in.tmp as the input file. It produces Tektronix hex format output file named out.x.

- **Example 4**

  If you enter:

  ```
  rom430 -t test
  ```

  the object format converter uses test.obj as the input file. It produces an output file named test.tag in TI–tagged format.

10.8 Halt Conditions

Two situations cause the object format converter to abort execution:

1) If any of the specified files cannot be opened, the object format converter prints the message *Input COFF file cannot be opened* and aborts.
2) If you supply the utility with the name of an invalid object file, the object format converter prints the message *Corrupt input file* and aborts.

Topics

Examples

Figures

Tables

20 Common Object File Format

The MSP430 assembler and linker create object files that are in common object file format (COFF). COFF is an implementation of an object file format of the same name that was developed by AT&T for use on UNIX–based systems. This object file format is used because it encourages modular programming and provides more powerful and flexible methods for managing code segments and target system memory.

One of the basic COFF concepts is *sections.* If you understand section operation, you will be able to use the assembly language tools more efficiently.

This appendix contains technical details about COFF object file structure. Much of this information pertains to the symbolic debugging information that is produced by high level programming languages. The main purpose of this appendix is to provide supplementary information for those of you who are interested in the internal format of COFF object files.

20.1 How the COFF File Is Structured

The elements of a COFF object file describe the file's sections and symbolic debugging information. These elements include:

- a file header.
- optional header information.
- a table of section headers.
- raw data for each initialized section.
- relocation information for each initialized section.
- line number entries for each initialized section.
- a symbol table.
- a string table.

The assembler and linker produce object files with the same COFF structure; however, a program that is linked for the final time does not usually contain relocation entries.

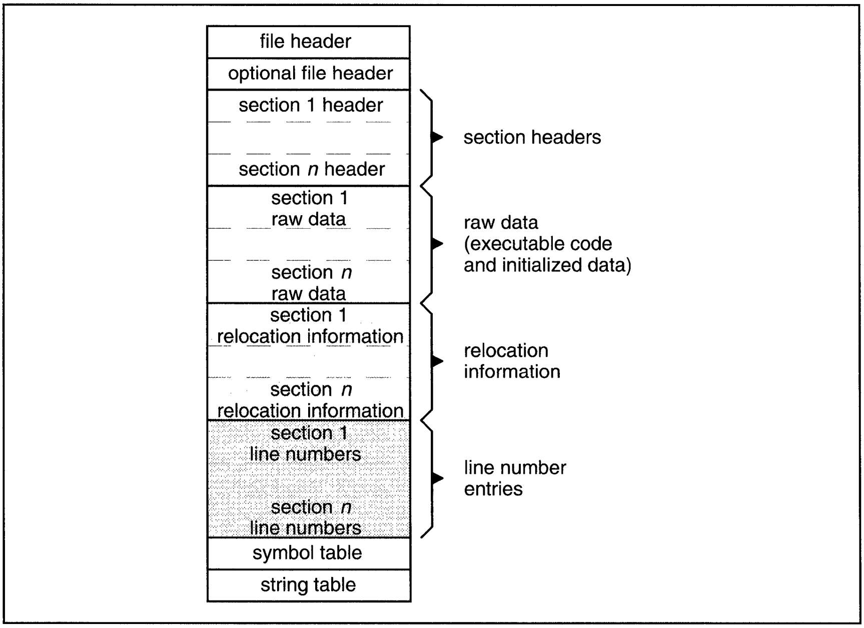


Figure 20.1: COFF File Structure

The following figure shows a typical example of a COFF object file that contains the three default sections, .text, .data, and .bss, and a named section (referred to as <named>). Although uninitialized sections have section headers, they have no raw data, relocation information, or line number entries. This is because the .bss and .usect directives simply reserve space for uninitialized data; uninitialized sections contain no actual code.

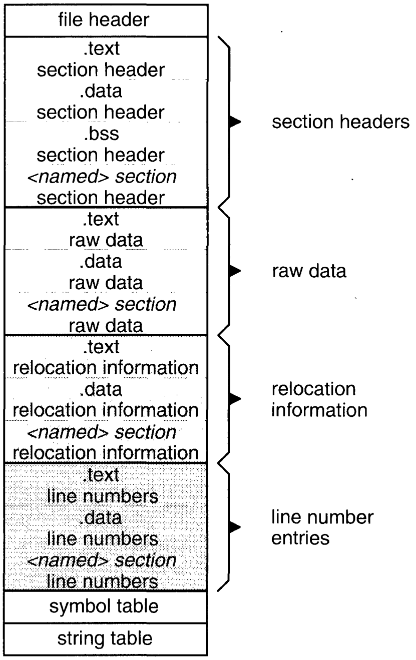


Figure 20.2: Sample COFF Object File

20.2 How the File Header Is Structured

The file header contains 20 bytes of information that describe the general format of an object file.

Byte Number	Type	Description
0-1	Unsigned short integer	Magic number, indicates that the file can be executed in a MSP430 system.
2-3	Unsigned short integer	Number of section headers.
4-7	Long integer	Time and date stamp, indicates when the file was created.
8-11	Long integer	File pointer, contains the symbol table's starting address.
12-15	Long integer	Number of entries in the symbol table.
16-17	Unsigned short integer	Number of bytes in the optional header. This field is either 0 or 28; if it is 0, there is no optional file header.
18-19	Unsigned short integer	Flags (see Table A-2).

Table 20.1: File Header Contents

The following table lists the flags that can appear in bytes 18 and 19 of the file header. Any number and combination of these flags can be set at the same time (for example, if bytes 18 and 19 are set to 0003h, F_RELFLG and F_EXEC are both set.)

Mnemonic	Flag	Description
F_RELFLG	0001h	Relocation information was stripped from the file.
F_EXEC	0002h	The file is relocatable (it contains no unresolved external references).
F_LNNO	0004h	Line numbers were stripped from the file.
F_LSYMS	0008h	Local symbols were stripped from the file.
F_BENDIAN	0200h	The file has the byte ordering used by MSP430 devices (most significant byte first).

Table 20.2: File Header Flags (Bytes 18 and 19)

20.3 Optional File Header Format

The linker creates the optional file header and uses it to perform relocation at download time. Partially linked files do not contain optional file headers.

Byte Number	Type	Description
0-1	Short integer	Magic number
2-3	Short integer	Version stamp
4-7	Long integer	Size (in bytes) of executable code
8-11	Long integer	Size (in bytes) of initialized data
12-15	Long integer	Size (in bytes) of uninitialized data
16-19	Long integer	Entry point
20-23	Long integer	Beginning address of executable code
24-27	Long integer	Beginning address of initialized data

Table 20.3: Optional File Header Contents

20.4 How Section Headers Are Structured

COFF object files contain a table of section headers that define where each section begins in the object file. Each section has its own section header.

Byte Number	Type	Description
0-7	Character	Eight–character section name, padded with nulls
8-11	Long integer	Section's physical address
12-15	Long integer	Section's virtual address
16-19	Long integer	Section size in words
20-23	Long integer	File pointer to raw data
24-27	Long integer	File pointer to relocation entries
28-31	Long integer	File pointer to line number entries
32-33	Unsigned short integer	Number of relocation entries
34-35	Unsigned short integer	Number of line number entries
36-37	Unsigned short integer	Flags (see Table A-5)
38	Character	Reserved
39	Unsigned Character	Memory page number

Table 20.4: Section Header Contents

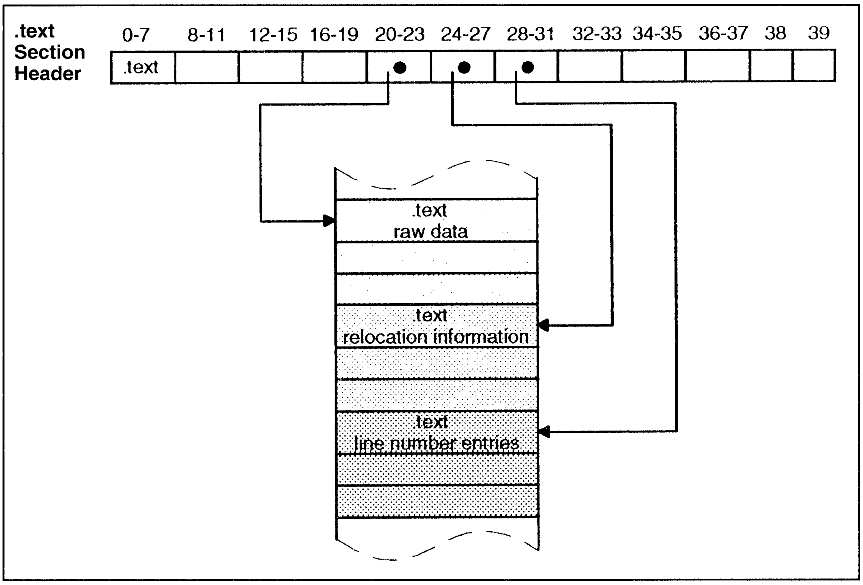
Mnemonic	Flag	Description
STYP_REG	0000h	Regular section (allocated, relocated, loaded)
STYP_DSECT	0001h	Dummy section (relocated, not allocated, not loaded)
STYP_NOLOAD	0002h	Noload section (allocated, relocated, not loaded)
STYP_GROUP	0004h	Grouped section (formed from several input sections)
STYP_PAD	0008h	Padding section (loaded, not allocated, not relocated)
STYP_COPY	0010h	Copy section (relocated, loaded, but not allocated; relocation and line number entries are processed normally)
STYP_TEXT	0020h	Section contains executable code
STYP_DATA	0040h	Section contains initialized data
STYP_BSS	0080h	Section contains uninitialized data

Note: The term *loaded* means that the raw data for this section appears in the object file.

Table 20.5: Section Header Flags (Bytes 36 and 37)

The flags listed in the last table can be combined; for example, if the flag's word is set to 024h, both STYP_GROUP and STYP_TEXT are set.

The example illustrates how the pointers in a section header would point to the various elements in an object file that are associated with the .text section.



Example 20.1: Section Header Pointers for the .text Section

As Figure A-2 , page A–3, shows, uninitialized sections (created with the .bss and .usect directives) vary from this format. Although uninitialized sections have section headers, they have no raw data, no relocation information, and no line number information; also, they occupy no actual space in the object file. Therefore, the number of relocation entries, the number of line number entries, and the file pointers are 0 for an uninitialized section. The header of an uninitialized section simply tells the linker how much space for variables it should reserve in the memory map.

20.5 Structuring Relocation Information

A COFF object file has one relocation entry for each relocatable reference. The assembler automatically generates relocation entries. The linker reads the relocation entries as it reads each input section and performs relocation. The relocation entries determine how references within each input section are treated.

The relocation information entries use the 10–byte format shown in the table.

Byte Number	Type	Description
0-3	Long integer	Virtual address of the reference
4-5	Unsigned short integer	Symbol table index (0-65535)
6-7	Unsigned short integer	Reserved
8-9	Unsigned short integer	Relocation type (see Table A-7)

Table 20.6: Relocation Entry Contents

- The **virtual address** is the symbol's address in the current section *before* relocation; it specifies *where* a relocation must occur. (This is the address of the field in the object code that must be patched.)

  Here's an example of code that generates a relocation entry:

  ```
  2                          .global  X
  3     0000      !40300000  br       #X
  ```

  In this example, the virtual address of the relocatable field is 0001.

- The **symbol table index** is the index of the referenced symbol. In the preceding example, this field would contain the index of X in the symbol table. The amount of the relocation is the difference between the symbol's current address in the section and its assembly–time address. The relocatable field must be relocated by the same amount as the referenced symbol. In the example, X has a value of 0 before relocation. Suppose X is relocated to address 2000h. This is the relocation amount (2000h - 0 = 2000h), so the relocation field at address 1 is patched by adding 2000h to it.

  You can determine a symbol's relocated address if you know which section it is defined in. For example, if X is defined in .data and .data is relocated by 2000h, X is relocated by 2000h.

  If the symbol table index in a relocation entry is -1 (0FFFFh), this is called an *internal relocation.* In this case, the relocation amount is simply the amount by which the current section is being relocated.

- The **relocation type** specifies the size of the field to be patched and describes how the patched value should be calculated. The type field depends on the addressing mode that was used to generate the relocatable reference. In the preceding example, the actual address of the referenced symbol (X) will be placed in a 16–bit field in the object code. This is a 16–bit direct relocation, so the relocation type is R_RELWORD.

Mnemonic	Flag	Relocation Type
R_ABS	0000h	No relocation
R_RELBYTE	000Fh	8–bit direct reference to symbol's address
R_RELWORD	0010h	16–bit direct reference to symbol's address
R_HIWORD	0031h	8–bit reference to MSB of word

Table 20.7: Relocation Types (Bytes 8 and 9)

20.6 How the Line Number Table Is Structured

The object file contains a table of line number entries that are useful for symbolic debugging. When the C compiler produces several lines of assembly language code, it creates a line–number entry that maps these lines back to the original line of C source code that generated them. Each single line number entry contains 6 bytes of information.

Byte Number	Type	Description
0-3	Long integer	This entry may have one of two values: 1) If it is the first entry in a block of line–number entries, it points to a symbol entry in the symbol table. 2) If it is not the first entry in a block, it is the physical address of the line indicated by bytes 4-5.
4-5	Unsigned short integer	This entry may have one of two values: 1) If this field is 0, this is the first line of a function entry. 2) If this field is *not* 0, this is the line number of a line of C source code.

Table 20.8: Line Number Entry Format

The figure shows how line number entries are grouped into blocks.

Symbol Index 1	**0**
physical address	line number
physical address	line number
Symbol Index*n*	**0**
physical address	line number
physical address	line number

Figure 20.3: Line Number Blocks

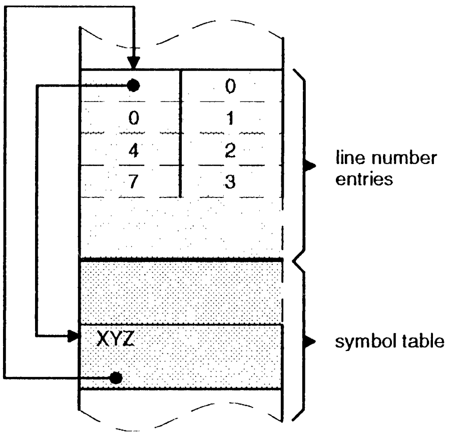
As the figure shows, each entry is divided into halves:

- For the *first line* of a function,
  - Bytes 0-3 point to the name of a symbol or a function in the symbol table.
  - Bytes 4-5 contain a 0, which indicates the beginning of a block.

- For the *remaining lines* in a function,
  - Bytes 0-3 show the physical address (the number of bytes created by a line of C source).
  - Bytes 4-5 show the address of the original C source, relative to its appearance in the C source program.

The line entry table can contain many of these blocks.

The following example illustrates the line number entries for a function named XYZ. As shown, the function name is entered as a symbol in the symbol table. The first portion on XYZ's block of line number entries points to the function name in the symbol table. Assume that the original function in the C source contained three lines of code. The first line of code produces 4 bytes of assembly language code, the second line produces 3 bytes, and the third line produces 10 bytes.



Example 20.2: Line Number Entries

(Note that the symbol table entry for XYZ has a field that points back to the beginning of the line number block.)

Because line numbers are not often needed, the linker provides an option (-s) that strips line number information from the object file; this provides a more compact object module.

20.7 Symbol Table Structure and Content

The order of symbols in the symbol table is very important.

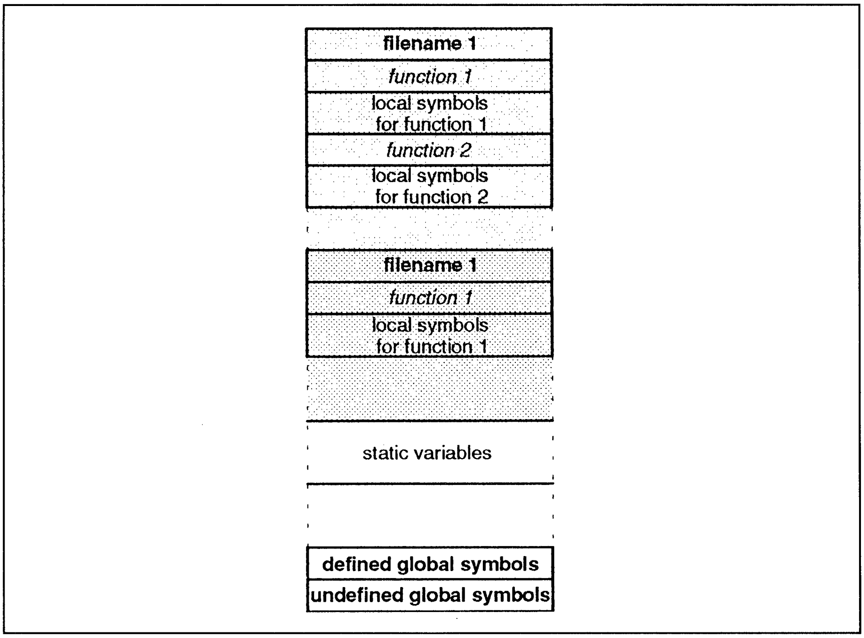


Figure 20.4: Symbol Table Contents

Static variables refer to symbols defined in C that have storage class static outside any function. If you have several modules that use symbols with the same name, making them static confines the scope of each symbol to the module that defines it (this eliminates multiple–definition conflicts).

The entry for each symbol in the symbol table contains the symbol's:

- Name (or an offset into the string table).
- Type.
- Value.
- Section it was defined in.
- Storage class.
- Basic type (integer, character, etc.).

- Derived type (array, structure, etc.).
- Dimensions.
- Line number of the source code that defined the symbol.

Section names are also defined in the symbol table.

All symbol entries, regardless of the symbol's class and type, have the same format in the symbol table. Each symbol table entry contains the 18 bytes of information listed in the next table. Each symbol may also have an 18–byte auxiliary entry; the special symbols listed in the table after next always have an auxiliary entry. Some symbols may not have all the characteristics listed above; if a particular field is not set, it is set to null.

Byte Number	Type	Description
0-7	Character	This field contains one of the following: 1) An 8–character symbol name, padded with nulls 2) An offset into the string table if the symbol name is longer than 8 characters
8-11	Long integer	Symbol value; storage class dependent
12-13	Short integer	Section number of the symbol
14-15	Unsigned short integer	Basic and derived type specification
16	Character	Storage class of the symbol
17	Character	Number of auxiliary entries (always 0 or 1)

Table 20.9: Symbol Table Entry Contents

20.7.1 Special Symbols Used in the Symbol Table

The symbol table contains some special symbols that are generated by the compiler, assembler, and linker. Each special symbol contains ordinary symbol table information and an auxiliary entry.

Symbol	Description
.file	File name
.text	Address of the .text section
.data	Address of the .data section
.bss	Address of the .bss section
.bb	Address of the beginning of a block
.eb	Address of the end of a block
.bf	Address of the beginning of a function
.ef	Address of the end of a function
.target	Pointer to a structure or union that is returned by a function
.*n*fake	Dummy tag name for a structure, union, or enumeration
.eos	End of a structure, union, or enumeration
etext	Next available address after the end of the .text output section
edata	Next available address after the end of the .data output section
end	Next available address after the end of the .bss output section

Table 20.10: Special Symbols in the Symbol Table

Several of these symbols appear in pairs:

- .eb indicate the beginning and end of a block.
- .bf/.ef indicate the beginning and end of a function.
- *n*fake/.eos name and define the limits of structures, unions, and enumerations that were not named. The .eos symbol is also paired with named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name, the compiler assigns it a name so that it can be entered into the symbol table. These names are of the form *n*fake, where *n* is an integer. The compiler begins numbering these symbol names at 0.

Symbols and Blocks

In C, a block is a compound statement that begins and ends with braces. A block always contains symbols. The symbol definitions for any particular block are grouped together in the symbol table and are delineated by the .bb/.eb special symbols. Note that blocks can be nested in C, and their symbol table entries can also be nested correspondingly. The following figure shows how block symbols are grouped in the symbol table.

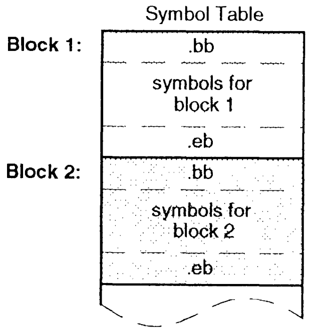


Figure 20.5: Symbols for Blocks

Symbols and Functions

The symbol definitions for a function appear in the symbol table as a group, delineated by .bf/.ef special symbols. The symbol table entry for the function name precedes the .bf special symbol. The next figure shows the format of symbol table entries for a function.

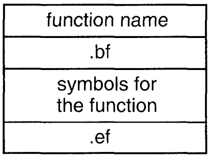


Figure 20.6: Symbols for Functions

If a function returns a structure or union, a symbol table entry for the special symbol .target will appear between the entries for the function name and the .bf special symbol.

20.7.2 Symbol Name Format

The first 8 bytes of a symbol table entry (bytes 0-7) indicate a symbol's name:

- If the symbol name is 8 characters or less, this field has type *character*. The name is padded with nulls (if necessary) and stored in bytes 0-7.
- If the symbol name is greater than 8 characters, this field is treated as two long integers. The entire symbol name is stored in the string table. Bytes 0-3 contain 0, and bytes 4-7 are an offset into the string table.

20.7.3 String Table Structure

Symbol names that are longer than eight characters are stored in the string table. The field in the symbol table entry that would normally contain the symbol's name contains, instead, a pointer to the symbol's name in the string table. Names are stored contiguously in the string table, delimited by a null byte. The first four bytes of the string table contain the size of the string table in bytes; thus, offsets into the string table are greater than or equal to four.

The next figure shows an example of a string table that contains two symbol names, Adaptive-Filter and Fourier-Transform. The index in the string table is 4 for Adaptive-Filter and 20 for Fourier-Transform.

38			
‘A’	‘d’	‘a’	‘p’
‘t’	‘i’	‘v’	‘e’
‘-’	‘F’	‘i’	‘l’
‘t’	‘e’	‘r’	‘\0’
‘F’	‘o’	‘u’	‘r’
‘i’	‘e’	‘r’	‘-’
‘T’	‘r’	‘a’	‘n’
‘s’	‘f’	‘o’	‘r’
‘m’	‘\0’		

Figure 20.7: Sample String Table

20.7.4 Storage Classes

Byte 16 of the symbol table entry indicates the storage class of the symbol. Storage classes refer to the method in which the C compiler accesses a symbol.

Mnemonic	Value	Storage Class	Mnemonic	Value	Storage Class
C_NULL	0	No storage class	C_USTATIC	14	Undefined static
C_AUTO	1	Automatic variable	C_ENTAG	15	Enumeration tag
C_EXT	2	External definition	C_MOE	16	Member of an enumeration
C_STAT	3	Static	C_REGPARM	17	Register parameter
C_REG	4	Register variable	C_FIELD	18	Bit field
C_EXTREF	5	External reference	C_UEXT	19	Tentative external definition
C_LABEL	6	Label	C_STATLAB	20	Static load time label
C_ULABEL	7	Undefined label	C_EXTLAB	21	External load time label
C_MOS	8	Member of a structure	C_BLOCK	100	Beginning or end of a block; used only for the .bb and .eb special symbols
C_ARG	9	Function argument	C_FCN	101	Beginning or end of a function; used only for the .bf and .ef special symbols
C_STRTAG	10	Structure tag	C_EOS	102	End of structure; used only for the .eos special symbol
C_MOU	11	Member of a union	C_FILE	103	Filename; used only for the .file special symbol
C_UNTAG	12	Union tag	C_LINE	104	Used only by utility programs
C_TPDEF	13	Type definition			

Table 20.11: Symbol Storage Classes

Some special symbols are restricted to certain storage classes.

Special Symbol	Restricted to This Storage Class	Special Symbol	Restricted to This Storage Class
.file	C_FILE	.eos	C_EOS
.bb	C_BLOCK	.text	C_STAT
.eb	C_BLOCK	.data	C_STAT
.bf	C_FCN	.bss	C_STAT
.ef	C_FCN		

Table 20.12: Special Symbols and Their Storage Classes

20.7.5 Symbol Values

Bytes 8-11 of a symbol table entry indicate a symbol's value. A symbol's value depends on the symbol's storage class; the table summarizes the storage classes and related values.

Storage Class	Value Description	Storage Class	Value Description
C_AUTO	Stack offset in bits	C_UNTAG	0
C_EXT	Relocatable address	C_TPDEF	0
C_STAT	Relocatable address	C_ENTAG	0
C_REG	Register number	C_MOE	Enumeration value
C_LABEL	Relocatable address	C_REGPARM	Register number
C_MOS	Offset in bits	C_FIELD	Bit displacement
C_ARG	Stack offset in bits	C_BLOCK	Relocatable address
C_STRTAG	0	C_FCN	Relocatable address
C_MOU	Offset in bits	C_FILE	0

Table 20.13: Symbol Values and Storage Classes

If a symbol's storage class is C_FILE, the symbol's value is a pointer to the next .file symbol. Thus, the .file symbols form a one–way linked list in the symbol table. When there are no more .file symbols, the final .file symbol points back to the first .file symbol in the symbol table.

The value of a relocatable symbol is its virtual address. When the linker relocates a section, the value of a relocatable symbol changes accordingly.

20.7.6 Section Number

Bytes 12-13 of a symbol table entry contain a number that indicates which section the symbol was defined in. The table lists these numbers and the sections they indicate.

Mnemonic	Section Number	Description
N_DEBUG	-2	Special symbolic debugging symbol
N_ABS	-1	Absolute symbol
N_UNDEF	0	Undefined external symbol
N_SCNUM	1	.text section (typical)
N_SCNUM	2	.data section (typical)
N_SCNUM	3	.bss section (typical)
N_SCNUM	1-32, 767	Section number of a named section, in the order in which the named sections are encountered

Table 20.14: Section Numbers

Note that if there were no .text, .data, or .bss sections, the numbering of named sections would begin with 1.

If a symbol has a section number of 0, -1, or -2, it is not defined in a section. A section number of -2 indicates a symbolic debugging symbol, which includes structure, union, and enumeration tag names, type definitions, and the filename. A section number of -1 indicates that the symbol has a value but is not relocatable. A section number of 0 indicates a relocatable external symbol that is not defined in the current file.

20.7.7 Type Entry

Bytes 14-15 of the symbol table entry define the symbol's type. Each symbol has one basic type and one to six derived types.

Here is the format for this 16–bit type entry:

	Derived Type 6	Derived Type 5	Derived Type 4	Derived Type 3	Derived Type 2	Derived Type 1	Basic Type
Size (in bits):	2	2	2	2	2	2	4

Bits 0-3 of the type field indicate the basic type.

Bits 4-15 of the type field are arranged as six 2–bit fields that can indicate 1 to 6 derived types.

Mnemonic	Value	Type	Mnemonic	Value	Type
T_VOID	0	Void type	T_STRUCT	8	Structure
T_SCHAR1	1	Character (explicitly signed)	T_UNION	9	Union
T_CHAR	2	Character (implicitly signed)	T_ENUM	10	Enumeration
T_SHORT	3	Short integer	T_LDOUBLE	11	Long Double Floating Point
T_INT	4	Integer	T_UCHAR	12	Unsigned character
T_LONG	5	Long integer	T_USHORT	13	Unsigned short integer
T_FLOAT	6	Floating point	T_UINT	14	Unsigned integer
T_DOUBLE	7	Double floating point	T_ULONG	15	Unsigned long integer

Table 20.15: Basic Types

Mnemonic	Value	Type	Mnemonic	Value	Type
DT_NON	0	No derived type	DT_FCN	2	Function
DT_PTR	1	Pointer	DT_ARY	3	Array

Table 20.16: Derived Types

An example of a symbol with several derived types would be a symbol with a type entry of 0000000011010011_2. This entry indicates that the symbol is an array of pointers to short integers.

20.7.8 Auxiliary Entries

Each symbol table entry may have **one** or **no** auxiliary entry. An auxiliary symbol table entry contains the same number of bytes as a symbol table entry (18), but the format of an auxiliary entry depends on the symbol's type and storage class. The following table summarizes these relationships.

Name	Storage Class	Type Entry		Auxiliary Entry Format
		Derived Type 1	Basic Type	
.file	C_FILE	DT_NON	T_VOID	Filename (see further tables)
.text, .data, .bss	C_STAT	DT_NON	T_VOID	Section (see further tables)
tagname	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_STRUCT T_UNION T_ENUM	Tag name (see further tables)
.eos	C_EOS	DT_NON	T_VOID	End of structure (see further tables)
fcname	C_EXT C_STAT	DT_FCN	(Any)	Function (see further tables)
arrname	(See note 2)	DT_ARY	(See note 1)	Array (see further tables)
.bb, .eb	C_BLOCK	DT_NON	T_VOID	Beginning and end of a block (see further tables)
.bf, .ef	C_FCN	DT_NON	T_VOID	Beginning and end of a function (see further tables)
Name related to a structure, union, or enumeration	(See note 2)	DT_PTR DT_ARR DT_NON	T_STRUCT T_UNION T_ENUM	Name related to a structure, union, or enumeration (see further tables)

Notes: 1) Any except T_VOID
2) C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF, C_EXT

Table 20.17: Auxiliary Symbol Table Entries Format

In this table, *tagname* refers to any symbol name (including the special symbol *n*fake). *Fcname* and *arrname* refer to any symbol name.

A symbol that satisfies more than one condition should have a union format in its auxiliary entry. A symbol that satisfies none of these conditions should not have an auxiliary entry.

Filenames

Each of the auxiliary table entries for a filename contains a 14–character file name in bytes 0-13. Bytes 14-17 are unused.

Byte Number	Type	Description
0-13	Character	File name
14-17	—	Unused

Table 20.18: Filename Format for Auxiliary Table Entries

Sections

Byte Number	Type	Description
0-3	Long integer	Section length
4-6	Unsigned short integer	Number of relocation entries
7-8	Unsigned short integer	Number of line number entries
9-17	—	Not used (zero filled)

Table 20.19: Section Format for Auxiliary Table Entries

Tag Names

Byte Number	Type	Description
0-3	—	Unused (zero filled)
4-7	Unsigned long integer	Size of structure, union, or enumeration
8-11	—	Unused (zero filled)
12-15	Long integer	Index of next entry beyond this structure, union, or enumeration
16-17	—	Unused (zero filled)

Table 20.20: Tag Name Format for Auxiliary Table Entries

End of Structure

Byte Number	Type	Description
0-3	Long integer	Tag index
4-7	Unsigned long integer	Size of structure, union, or enumeration
8-17	—	Unused (zero filled)

Table 20.21: End–of–Structure Format for Auxiliary Table Entries

Functions

Byte Number	Type	Description
0-3	Long integer	Tag index
4-7	Long integer	Size of function (in bits)
8-11	Long integer	File pointer to line number
12-15	Long integer	Index of next entry beyond this function
16-17	—	Unused (zero filled)

Table 20.22: Function Format for Auxiliary Table Entries

Arrays

Byte Number	Type	Description
0-3	Long integer	Tag index
4-7	Unsigned long integer	Size of array
8-9	Unsigned short integer	First dimension
10-11	Unsigned short integer	Second dimension
12-13	Unsigned short integer	Third dimension
14-15	Unsigned short integer	Fourth dimension
16-17	—	Unused (zero filled)

Table 20.23: Array Format for Auxiliary Table Entries

End of Blocks and Functions

Byte Number	Type	Description
0-3	—	Unused (zero filled)
4-5	Unsigned short integer	C source line number
6-17	—	Unused (zero filled)

Table 20.24: End–of–Blocks/Functions Format for Auxiliary Table Entries

Beginning of Blocks and Functions

Byte Number	Type	Description
0-3	Unsigned long integer	Register save mask
4-5	Unsigned short integer	C source line number of block begin
6-7	Unsigned short integer	Number line entries for function
8-11	Unsigned long integer	Size of local frame for function
12-15	Long integer	Index of next entry past this block
16-17	—	Unused (zero filled)

Table 20.25: Beginning–of–Blocks/Functions Format for Auxiliary Table

Names Related to Structures, Unions, and Enumerations

Byte Number	Type	Description
0-3	Long integer	Tag index
4-7	Unsigned long integer	Size of the structure, union, or enumeration
8-17	—	Unused (zero filled)

Table 20.26: Structure, Union, and Enumeration Names Format for Auxiliary Table Entries

Topics

21 Symbolic Debugging Directives

The MSP430 fixed–point assembler supports several directives that a high level programming language can use for symbolic debugging:

- The **.sym** directive defines a global variable, a local variable, or a function. Several parameters allow you to associate various debugging information with the symbol 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 function.
- The **.block** and **.endblock** directives specify the bounds of blocks.
- The **.file** directive defines a symbol in the symbol table that identifies the current source file name.
- The **.line** directive identifies the line number of a source statement.

These symbolic debugging directives are not usually listed in the assembly language file that the compiler creates. If you want them to be listed, invoke the compiler shell with the -g option, as shown below:

```
cl430 -g input file
```

This appendix contains an alphabetical directory of the symbolic debugging directives. Each directive contains an example of C source and the resulting assembly language code.

Syntax

.block [*beginning line number*]

.endblock [*ending line number*]

Description

The .block and .endblock directives specify the beginning and end of a block. The line numbers are optional; they specify the location in the source file where the block is defined. Line numbers are relative to the beginning of the current function.

Note that block definitions can be nested. The assembler will detect improper block nesting.

Example

Here is an example of C source that defines a block and of the resulting assembly language code.

C source:

```
     .
     .
     .
{                    /*  Beginning of a block  */
     int  a,b;
     a = b;
}                    /*  End of a block        */
     .
     .
     .
```

Resulting assembly language code:

```
.block    4
.sym      _a,1,4,1,8
.sym      _b,2,4,1,8
.line 5
MOV   2(SP), 1(SP)
.endblock    6
```

Syntax **.file** *filename*

Description The .file directive allows a debugger to map locations in memory back to lines in a source file. The *filename* is the name of the file that contains the original C source program. The first 14 characters of the filename are significant; any pathname information is stripped away.

You can use the .file directive in assembly code to provide a name in the file and improve program readability.

Example Here's an example of the .file directive. The file named *text.c* contained the C source that produced this directive.

```
.file    "text.c"
```

Syntax

.func [*beginning line number*]

.endfunc [*ending line number*] [, *register save mask1*] [, *register save mask2*] [, *frame size*]

Description

The .func and .endfunc directives specify the beginning and end of a function. The line numbers are optional; they specify the location in the source file where the function is defined. The register save masks indicate which registers were saved by this function. If bit 0 of mask2 is 1, R0 was saved by the function; if bit 1 of mask2 is 1, R1 was saved; if bit 0 of mask1 is 1, R16 was saved; etc. The frame size parameter indicates how many bytes were reserved for the local frame of this function.

Note that function definitions cannot be nested.

Example

Here is an example of C source that defines a function and of the resulting assembly language code.

C source:

```
power(x, n)             /*  Beginning of a function
   */
int x,n;
{
      register int i, p;
      p = 1;
      for (i = 1; i <= n; ++i) p*= x;
      return p;         /*  End of function          */
}
```

Resulting assembly language code:

```
       .sym    _power,_power,36,2,0
       .global       _power
       .text
       .func  1
;*****************************************************
;* FUNCTION DEF : _power
;*****************************************************
_power:;
       INCW    #4,STK
       POP     A
       MOV     A,-2(STK)
       POP     A
       MOV     A,-3(STK)
       MOV     FP,-1,A
       MOV     A,-1(STK)
       MOV     FP,A
       MOV     A,@STK
       MOVW    STK,FP
       INCW    #2,STK
       MOV     R23,A
       MOV     A,-1(STK)
       MOV     R24,A
       MOV     A,@STK
       .sym    _x,-4,4,9,8
       .sym    _n,-5,4,9,8
       .sym    _i,23,4,4,8
       .sym    _p,24,4,4,8
       .line   3
       .line   5
       MOV     #01h,R24
       .line   6
       MOV     #01h,R23
       JMP     L2
L1:
       MOV     -4(FP),A
       MPY     R24,A
       MOV     B,R24
       INC     R23
L2:
       MOV     -5(FP),A
       CMP     R23,A
       JGE     L1
       .line   7
       MOV     R24,R8
EPIO_1:
       .line   8
       MOV     @STK,A
       MOV     A,R24
       MOV     -1(STK),A
       MOV     A,R23
       MOVW    FP,STK
       MOV     @STK,A
       MOV     A,FP
       MOV     -1(STK),A
       MOV     A,FP-1
       MOV     -2(STK),A
       MOV     A,B
       MOV     -3(STK),A
       INCW    #-4,STK
       BR      @R1
       .endfunc  8,00180H,00000H,0
       .end
```

Syntax **.line** *line number [, address]*

Description The .line directive creates a line number entry in the object file. Line number entries are used in symbolic debugging to associate addresses in the object code with the lines in the source code that generated them.

The .line directive has two operands:

- *Line number* indicates the line of the source that generated a portion of code. Line numbers are relative to the beginning of the current function. This is a required parameter.
- *Address* is an expression that is the address associated with the line number. This is an optional parameter; if you don't specify an address, the assembler will use the current SPC value.

Example The .line directive is followed by the assembly language source statements that are generated by the indicated line of C source. For example, assume that the lines of C source below are lines 5 and 6 in the original C source; lines 5 and 6 produce the assembly language source statements that are shown below.

C source:

```
for (p = 1; i = 1; i <= n; ++i) p*=x
return p;
```

Resulting assembly language code:

```
   .line 5
   MOV #01h, R24
   MOV #01h, R23
   JMP L2
L1:
   MOV -4(FP), A
   MPY R24, A
   MOV B, R24
   INC R23
L2:
   MOV -5(FP), A
   CMP R23, A
   JGE L1
   .line 6
   MOV R24, R8
```

Syntax **.member** *name, value [, type, storage class, size, tag, dims]*

Description The .member directive defines a member of a structure, union, or enumeration. It is valid only when it appears in a structure, union, or enumeration definition.

- *Name* is the name of the member that is put in the symbol table. The first 32 characters of the name are significant.
- *Value* is the value associated with the member. Any legal expression (absolute or relocatable) is acceptable.
- *Type* is the type of the member. Appendix A contains more information about types.
- *Storage class* is the storage class of the member. Appendix A contains more information about storage classes.
- *Size* is the number of bits of memory required to contain this member.
- *Tag* is the name of the type (if any) or structure of which this member is a type. This name **must** have been previously declared by a .stag, .etag, or .utag directive.
- *Dims* may be one to four expressions separated by commas. This allows up to four dimensions to be specified for the member.

The order of parameters is significant. *Name* and *value* are required parameters. All other parameters may be omitted or empty (adjacent commas indicate an empty entry). This allows you to skip a parameter and specify a parameter that occurs later in the list. Operands that are omitted or empty assume a null value.

Example Here is an example of a C structure definition and the corresponding assembly language statements:

C source:

```
struct doc {
     char title;
     char group;
     int  job_number;
} doc_info;
```

Resulting assembly language code:

```
     .stag   doc,24
     .member  _title,0,2,8,8
     .member  _group,8,2,8,8
     .member _job_number,16,4,8,8
     .eos
```

Syntax

.stag *name [, size]*
member definitions
.eos

.etag *name [, size]*
member definitions
.eos

.utag *name [, size]*
member definitions
.eos

Description

The .stag directive begins a structure definition. The .etag directive begins an enumeration definition. The .utag directive begins a union definition. The .eos directive ends a structure, enumeration, or union definition.

Name is the name of the structure, enumeration, or union. The first 32 characters of the name are significant. This is a required parameter.

Size is the number of bits the structure, enumeration, or union occupies in memory. This is an optional parameter; if omitted, the size is unspecified.

The .stag, .etag, or.utag directive should be followed by a number of .member directives, which define members in the structure. The .member directive is the only directive that can appear inside a structure, enumeration, or union definition.

The assembler does not allow nested structures, enumerations, or unions. A C compiler "unwinds" nested structures by defining them separately and then referencing them from the structure they are referenced in.

Example 1

Here is an example of a structure definition.

C source:

```
struct doc
{
    char  title;
    char  group;
    int   job_number;
} doc_info;
```

Resulting assembly language code:

```
.stag   _doc,24
.member _title,0,2,8,8
.member _group,8,2,8,8
.member _job_number,16,4,8,8
.eos
```

Example 2 Here is an example of a union definition.

C source:

```
union u_tag {
    int   val1;
    float val2;
    char  valc;
} valu;
```

Resulting assembly language code:

```
        .utag   _u_tag,24
        .member _val1,0,4,11,8
        .member _val2,0,6,11,24
        .member _valc,0,2,11,8
        .eos
```

Examp Here is an example of an enumeration definition.

C Source:

```
{
    enum o_ty { reg_1, reg_2, result } optypes;
}
```

Resulting assembly language code:

```
        .etag   _o_ty,8
        .member _reg_1,0,4,16,8
        .member _reg_2,1,4,16,8
        .member _result,2,4,16,8
        .eos
```

Syntax

.sym *name, value [, type, storage class, size, tag, dims]*

Description

The .sym directive specifies symbolic debug information about a global variable, local variable, or a function.

- *Name* is the name of the variable that is put in the object symbol table. The first 32 characters of the name are significant.
- *Value* is the value associated with the variable. Any legal expression (absolute or relocatable) is acceptable.
- *Type?* is the type of the variable. Appendix A contains more information about types.
- *Storage class?* is the storage class of the variable. Appendix A contains more information about storage classes.
- *Size?* is the number of bits of memory required to contain this variable.
- *Tag* is the name of the type (if any) or structure of which this variable is a type. This name **must** have been previously declared by a .stag, .etag, or .utag directive.
- *Dims* may be up to four expressions separated by commas. This allows up to four dimensions to be specified for the variable.

The order of parameters is significant. *Name* and *value* are required parameters. All other parameters may be omitted or empty (adjacent commas indicate an empty entry). This allows you to skip a parameter and specify a parameter that occurs later in the list. Operands that are omitted or empty assume a null value.

Example

These lines of C source produce the .sym directives shown below:

C source:

```
struct s { int member1, member2; } str;
int ext;
int array[5][10];
long *ptr;
int strcmp();

main(arg1,arg2)
   int arg1;
   char *arg2;
{
   register r1;
}
```

Resulting assembly language code:

```
.sym     _str,_str,8,2,16,_s
.sym     _ext,_ext,4,2,8
.sym     _array,_array,244,2,400,,5,10
.sym     _ptr,_ptr,21,2,16
.sym     _main,_main,36,2,0
.sym     _arg1,-4,4,9,8
.sym     _arg2,-6,18,9,16
.sym     _r1,23,4,4,8
```

Topics

22 Assembler Error Messages

The assembler issues several types of error messages:

- Fatal
- Nonfatal
- Macro

When the assembler completes its second pass, it reports any errors that it encountered during the assembly. It also prints these errors in the listing file (if one is created); an error is printed following the source line that incurred it.

This appendix discusses the three types of assembler error messages; they are listed in alphabetical order. Most errors are fatal errors; if an error is not fatal or if it is a macro error, this is noted in the list.

absolute value required: A relocatable symbol was used where an absolute symbol was expected.

a component of the expression is invalid

address required: The operand of the flagged directive must be an address

an identifier in the expression is invalid

argument must be character constant

bad indirect address

bad macro library format

.break encountered outside loop block

cannot equate an external to an external

cannot open library: A library name specified with the .mlib directive does not exist or is already being used.

cannot redefine register: Register names cannot be used as labels.

character constant overflows a word

close ()) missing: Mismatched parentheses.

close (]) missing: Mismatched brackets.

close quote missing: All strings must be enclosed in quotes.

comma missing: The assembler expected a comma but did not find one. This usually means that more operands were expected.

conditional block nesting level exceeded

conflicts with previous section definition

copy file open error: A file specified by a .copy directive does not exist or is already being used.

directive only valid if (-a) option use: The .setsect and .setsym directives can be used only if the -a (absolute list) option is specified.

divide by zero: An expression or well–defined expression contains invalid division.

duplicate definition: The symbol appears as an operand of a REF statement, as well as in the the label field of the source, or the symbol appears more than once in the label field of the source.

duplicate definition of a structure component

.else or .elseif needs corresponding .if: An .else or .elseif directive was not preceded by an .if directive.

empty structure

expression changed values due to jump expansion: An expression is dependent on the amount of code between 2 labels. If the assembler expands a jump in the code between these 2 labels, then this expression will evaluate to different values in pass1 and pass2. Between the 2 labels, you will need to manually expand any jumps in your source code which were automatically expanded by the assembler.

expression not terminated properly

expression out of bounds

filename missing: The specified filename cannot be found.

floating-point expression not allowed

floating-point number not valid in expression

illegal label: A label cannot be used for the second instruction of a parallel instruction pair.

illegal operation in expression

illegal structure definition

illegal structure member

illegal structure, union, or enumeration tag

illegal relative address: The label destination of a relative jump must be defined within the same section as the jump.

illegal symbolic address: Operand only valid in absolute address mode.

illegal use of local label: Local labels are not allowed in expressions.

invalid binary constant: The only valid binary integers are 0 and 1; the constant must be suffixed with b or B.

invalid bit number: You must specify a bit number between 0 and 7.

invalid decimal constant: The only valid decimal integers are 0-9.

invalid expression: This may indicate invalid use of a relocatable symbol in arithmetic.

invalid floating–point constant

invalid octal constant: The only valid octal digits are the integers are 0-8; the constant must be suffixed with q or Q.

invalid opcode: The command field of the source record has an entry that is not a defined instruction, directive, or macro name.

invalid operand or operand combination

invalid option: An option specified by the .option directive is invalid.

invalid subscript or index

invalid symbol qualifier

invalid trap number: Trap numbers must be absolute values between 0 and 15.

label required: The flagged directive must have a label.

library not in archive format: A file specified with an .mlib directive is not an archive file.

local label multiply defined in block

local label not defined in block

local macro variable is not a valid symbol

macro parameter is not a valid symbol

maximum macro nesting level exceeded

maximum number of copy files exceeded

.mexit directive encountered outside macro

missing .endif directive

missing .endloop directive

missing .endm directive

missing macro name

missing structure tag

no include/copy files in macro or loop blocks

no parameters for macro arguments

no relative jumps to symbols not in current section: Relative jumps to load–time addresses defined with the .label directive are not allowed.

offset must point to even (word) address

open “(“ expected

operand missing: An operand must be supplied.

operand must be an immediate value

pass1/pass2 operand conflict: A symbol in the symbol table did not have the same value in pass 1 and pass 2.

positive value required

redefinition of local substitution symbol

register symbol used before definition: Equating a symbol to a register must be done before first symbol use.

relative jumps to externals are not allowed

string required: You must supply a string that is enclosed in double quotes.

substitution symbol stack overflow

substitution symbol string too long

subtraction of labels not allowed: Subtraction of labels or relationals involving the amount of code between labels is not allowed in expressions used in some contexts.

symbol required: The .global directive requires a symbol as an operand.

symbol used in both REF and DEF: A REFed symbol is already defined.

syntax error

target address not word aligned

too many local substitution symbols

unbalanced symbol table entries: For .block and .func directives.

undefined structure member

undefined structure tag

undefined substitution symbol

undefined symbol: An undefined symbol was used where a well–defined expression is required.

underflow in floating–point constant: Floating–point value is too small to represent.

unexpected .endif encountered: An .endif directive was not preceded by a .loop directive.

unexpected .endloop encountered: An .endloop directive was not preceded by a .loop directive.

unexpected .endm directive encountered

unexpected .endstruct directive encountered: An .endstruct directive was not preceded by a .struct directive.

value is out of range

.var directive encountered outside macro

version number changed

warning — block open at end of file

warning — function .sym required before .func

warning — immediate operand not absolute

warning — line truncated

warning — register converted to immediate

warning — string length exceeds maximum limit

warning — symbol truncated: The maximum length for a symbol is eight characters. The assembler ignores the extra characters.

warning — trailing operand(s): The assembler found fewer or more operands than expected in the flagged instruction.

warning — value out of range

warning — value truncated: The expression given was too large to fit within the instruction opcode or the required number of bits.

Topics

23 Linker Error Messages

The linker issues several types of error messages:

- Syntax and command errors
- Allocation errors
- I/O errors

This appendix discusses the three types of errors; they are listed alphabetically within each category. In these listings, the symbol (...) represents the name of an object that the linker is attempting to interact with when an error occurs.

- **Syntax/Command Errors**

  These errors are caused by incorrect use of linker directives, misuse of an input expression, or invalid options. Check the syntax of all expressions, and check the input directives for accuracy. Review the various options you are using and check for conflicts.

  absolute symbol (...) being redefined: An absolute symbol cannot be redefined.

  adding name (...) to multiple output sections: The input section is mentioned twice in the SECTIONS directive.

  ALIGN illegal in this context: Alignment of a symbol can be performed only within a SECTIONS directive.

  attempt to decrement DOT: Statements such as **.-= value** are illegal. Assignments to **dot** can be used only to create holes.

  bad fill value: The fill value must be a 16–bit constant.

  binding address for (...) redefined: Only one binding value is allowed for each section.

  blocking for (...) redefined: Only one blocking value is allowed for each section.

  can't open filename: Specified filename cannot be opened for some reason; file doesn't exist, wrong file type, etc.

  cannot specify both binding and memory area for (...): The two are mutually exclusive. If you wish the code to be placed at a specific address, use binding only.

  cannot specify a page for a section within a GROUP

  command file nesting exceeded with file (...): Command file nesting is allowed up to 16 levels.

  -e flag does not specify a legal symbol name (...): The -e option requires a valid symbol name as an operand.

  entry point symbol (...) undefined: The symbol used with the -e option is not defined.

  errors in input - (...) not built: Previous errors prevent the creation of an output file.

  fill value for (...) redefined: Only one fill value is allowed per output section. Individual holes can be filled with different values with the section definition.

-i path too long (...): The maximum number of characters in an -i path is 256.

illegal input character: There is a control character or other unrecognized character in the command file.

illegal memory attributes for (...): The attributes must be some combination of R, W, I, and X.

illegal operator in expression: Review legal expression operators.

illegal option within SECTIONS: The -l (lowercase L) is the only option allowed within a SECTIONS directive.

invalid path specified with -i flag: The operand of the -i flag must be a valid file or pathname.

invalid value for -f flag: must be a 2–byte constant.

invalid value for -heap flag: must be a 2–byte constant.

invalid value for -stack flag: must be a 2–byte constant.

invalid value for -v flag: must be a constant.

length redefined for memory area (...): Each memory area in a MEMORY directive can have only one length.

-m flag does not specify a valid filename: You must specify a valid filename to write the output map file to.

memory area for (...) redefined: Only one named memory allocation is allowed for each output section.

memory page for (...) redefined: Only one page allocation is allowed for each section.

memory attributes redefined for (...): Only one set of memory attributes is allowed for each output section.

missing filename on -l; use -l <filename>: The -l (lowercase L) option requires the use of a filename operand.

misuse of DOT symbol in assignment instruction: The dot symbol cannot be used in assignment statements that are outside SECTIONS directives.

no input files: The linker cannot operate without at least one input COFF file.

-o flag does specify a valid file name : string

output file has no .bss section: This is a warning. This section is usually present in a COFF file. There is no real requirement for it to be present.

output file has no .data section: This is a warning. This section is usually present in a COFF file. There is no real requirement for it to be present.

output file has no .text section: This is a warning. This section is usually present in a COFF file. There is no real requirement for it to be present.

origin missing for memory area (...)

origin redefined for memory area (...)

-r incompatible with -s (-s ignored): Since the -s option strips the relocation information and -r requests a relocatable object file, these options are in conflict with each other.

section (...) not built: The most likely cause of this is a syntax error in the SECTIONS directive.

semicolon required after assignment: There is a syntax error in the command file.

statement ignored: Caused by a syntax error in an expression.

symbol referencing errors — (...) not built

symbol (...) from file (...) being redefined: A defined symbol cannot be redefined in an assignment statement.

too many arguments - use a command file: You are limited to ten arguments on a command line, or in response to prompts.

too many -i options, 7 allowed: Additional search directories can be specified with a C_DIR or A_DIR environment variable.

type flags for (...) redefined: Only one section type is allowed per section. Note that type COPY has all of the attributes of type DSECT, so DSECT need not be specified separately.

type flags not allowed for GROUP or UNION: Special section types apply to individual sections only.

-u does not specify a legal symbol name: The -u option must specify a legal symbol name that exists in one of the files that you are linking.

unexpected EOF(end of file): Syntax error in the linker command file.

undefined symbol in expression: An assignment statement contains an undefined symbol.

unrecognized option (...): Check the list of valid options.

zero or missing length for memory area (...): Each memory range defined with the MEMORY directive must have a nonzero length.

- **Allocation Errors**

  These error messages appear during the allocation phase of linking. They generally appear if a section or group does not fit at a certain address or if the MEMORY and SECTIONS directives conflict in some way. If you are using a linker command file, check that MEMORY and SECTIONS directives allow enough room to ensure that no sections overlap and that no sections are being placed in unconfigured memory.

  alignment for (...) must be a power of 2: Section alignment must be a power of 2.

  alignment for (...) redefined: Only one alignment is allowed for each section.

  binding address (...) for section (...) is outside all memory on page (...): Each section must fall within memory configured with the MEMORY directive.

  binding address (...) for section (...) overlays (...) at (...): Two sections overlap and cannot be allocated.

  binding address (...) incompatible with alignment for section (...): The section has an alignment requirement from a .align directive or previous link. The binding address violates this requirement.

  blocking for (...) must be a power of 2: Section blocking must be a power of 2.

  can't align a section within GROUP - (...) not aligned: The entire GROUP is treated as one unit, so the GROUP can be aligned or bound to an address, but the sections making up the GROUP cannot be handled individually.

  can't align within UNION - section (...) not aligned: The entire UNION is treated as one unit, so the UNION can be aligned or bound to an address, but the sections making up the UNION cannot be handled individually.

  can't allocate (...), size ... (page ...): A section can't be allocated, because no configured memory area exists that is large enough to hold it.

  load address for uninitialized section (...) ignored: Uninitialized sections have no load addresses—only run addresses.

  load address for UNION ignored: UNION refers only to the section's run address.

  load allocation required for uninitialized UNION member (...): UNIONs refer to runtime allocation only. You must specify the load address for all sections within a UNION separately.

  no allocation allowed for uninitialized UNION member: An uninitialized section with a UNION gets its run allocation from the UNION and has no load address, so no allocation is valid for the member.

  no allocation allowed with a GROUP-allocation for section (...) ignored: The entire group is treated as one unit, so the group can be aligned or bound to an address, but the sections making up the group cannot be handled individually.

  no load address specified for (...); using run address: If an initialized section has a run address, only the section is allocated to run and load at the same address.

  no run allocation allowed for union member (...): A UNION defines the run address for all of its members; therefore, individual run allocations are illegal.

output file (...) not executable: The output file created may have unresolved symbols or other problems stemming from other errors. This condition is not fatal.

PC–relative displacement overflow at address (...) in file (...): relocation of a PC–relative jump resulted in a jump displacement too large to encode in the instruction.

section (...) at (...) overlays at address (...): The two sections overlap and cannot be allocated.

section (...) enters unconfigured memory at address (...): A section can't be allocated because no configured memory area exists that is large enough to hold it.

section (...) not found: An input section specified in a SECTIONS directive was not found in the input file.

section (...) won't fit into configured memory: A section can't be allocated, because no configured memory area exists that is large enough to hold it.

undefined symbol (...) first referenced in file (...): Unless the -r option is used, the linker requires that all referenced symbols be defined. This condition prevents the creation of an executable output file.

- **I/O and Internal Overflow Errors:**

  The following error messages indicate that the input file is corrupt, nonexistent, or unreadable, or that the output file cannot be opened or written to. Messages in this category may also indicate that the linker is out of memory or table space. Make sure that the input file is in the correct directory and that the file system is not out of space. If the input file is corrupt, try reassembling it.

  cannot complete output file (...), write error: Usually means that the file system is out of space.

  cannot create output file (...): Usually indicates an illegal filename.

  can't find input file *filename*:

  can't open (...): The specified file does not exist.

  can't read (...)

  can't seek (...)

  can't write (...)

  can't create map file (...): Usually indicates an illegal filename.

  fail to copy (...)

  fail to read (...)

  fail to seek (...)

  fail to skip (...)

  fail to write (...)

  file (...) has no relocation information: You have attempted to relink a file that was not linked with -r.

  file (...) is of unknown type, magic number = (...): The binary input file is not a COFF file.

  illegal relocation type (...) found in section(s) of file (...): The binary file is corrupt.

  internal error (...): Indicates an internal error is in the linker.

  invalid archive size for file (...): The archive file is corrupt.

  I/O error on output file (...)

  library (...) member (...) has no relocation information

  line number entry found for absolute symbol: The input file is corrupt.

  making aux entry *filename* for symbol *n* out of sequence: The input file is corrupt.

  no string table in file *filename*: The input file is corrupt.

  no symbol map produced - not enough memory: This is a nonfatal condition that prevents the generation of the symbol list in the map file.

  overwriting aux entry filename of symbol n: The input file is corrupt.

  out of memory, aborting:

relocation entries out of order in section (...) of file (...): The input file is corrupt.

relocation symbol not found: index (...), section (...), file (...): The input file is corrupt.

seek to (...) failed

too few symbol names in string table for archive n: The archive file is corrupt.

Topics

24 ASCII Character Set

Base		Char	Base		Char	Base		Char	Base		Char
10	16		10	16		10	16		10	16	
0	00	**NULL**	32	20	**SP**	64	40	**@**	96	60	**‘**
1	01	**SOH**	33	21	**!**	65	41	**A**	97	61	**a**
2	02	**STX**	34	22	**”**	66	42	**B**	98	62	**b**
3	03	**ETX**	35	23	**#**	67	43	**C**	99	63	**c**
4	04	**EOT**	36	24	**$**	68	44	**D**	100	64	**d**
5	05	**ENQ**	37	25	**%**	69	45	**E**	101	65	**e**
6	06	**ACK**	38	26	**&**	70	46	**F**	102	66	**f**
7	07	**BEL**	39	27	**’**	71	47	**G**	103	67	**g**
8	08	**BS**	40	28	**(**	72	48	**H**	104	68	**h**
9	09	**HT**	41	29	**)**	73	49	**I**	105	69	**i**
10	0A	**LF**	42	2A	*	74	4A	**J**	106	6A	**j**
11	0B	**VT**	43	2B	**+**	75	4B	**K**	107	6B	**k**
12	0C	**FF**	44	2C	**,**	76	4C	**L**	108	6C	**l**
13	0D	**CR**	45	2D	**-**	77	4D	**M**	109	6D	**m**
14	0E	**SO**	46	2E	**.**	78	4E	**N**	110	6E	**n**
15	0F	**SI**	47	2F	**/**	79	4F	**O**	111	6F	**o**
16	10	**DLE**	48	30	**0**	80	50	**P**	112	70	**p**
17	11	**DC1**	49	31	**1**	81	51	**Q**	113	71	**q**
18	12	**DC2**	50	32	**2**	82	52	**R**	114	72	**r**
19	13	**DC3**	51	33	**3**	83	53	**S**	115	73	**s**
20	14	**DC4**	52	34	**4**	84	54	**T**	116	74	**t**
21	15	**NAK**	53	35	**5**	85	55	**U**	117	75	**u**
22	16	**SYN**	54	36	**6**	86	56	**V**	118	76	**v**
23	17	**ETB**	55	37	**7**	87	57	**W**	119	77	**w**
24	18	**CAN**	56	38	**8**	88	58	**X**	120	78	**x**
25	19	**EM**	57	39	**9**	89	59	**Y**	121	79	**y**
26	1A	**SUB**	58	3A	**:**	90	5A	**Z**	122	7A	**z**
27	1B	**ESC**	59	3B	**;**	91	5B	**[**	123	7B	**{**
28	1C	**FS**	60	3C	**<**	92	5C	\	124	7C	**|**
29	1D	**GS**	61	3D	**=**	93	5D	**]**	125	7D	**}**
30	1E	**RS**	62	3E	**>**	94	5E	**^**	126	7E	**~**
31	1F	**US**	63	3F	**?**	95	5F	_	127	7F	**DEL**

Topics

25 Glossary

A

absolute address: An address that is permanently assigned to a 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*–bit 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.

archive library: A collection of individual files that have been grouped into a single file.

archiver: A software program that allows you to collect several individual files into a single file called an archive library. The archiver also allows you to delete, extract, or replace members of the archive library, as well as to add new members.

assembler: A software program that creates a machine–language program from a source file that contains assembly language instructions, directives, and macro directives. 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 or .equ directive.

assignment statement: A statement that assigns a value to a variable.

auxiliary entry: The extra entry that a symbol may have in the symbol table and that contains additional information about the symbol (whether the symbol is a filename, a section name, a function name, etc.).

B

binding: A process in which you specify a distinct address for an output section or a symbol.

block: A set of declarations and statements that are grouped together with braces.

.bss: One of the default COFF sections. You can use the .bss directive to reserve a specified amount of space in the memory map that can later be used for storing data. The .bss section is uninitialized.

byte: A sequence of 8 adjacent bits operated upon as a unit.

C

C compiler: A program that translates C source statements into assembly language source statements.

command file: A file that contains linker options and names input files for the linker.

comment: A source statement (or portion of a source statement) that is used to document or improve readability of a source file. Comments are not compiled, assembled, or linked; they have no effect on the object file.

common object file format (COFF): An object file that promotes modular programming by supporting the concept of *sections.*

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 numeric value that can be used as an operand.

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.

D

.data: One of the default COFF 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.

directive: 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).

E

emulator: A hardware development system that emulates MSP430 operation.

entry point: The starting execution point in target memory.

executable module: An object file that has been linked and can be executed in a MSP430 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 in a different program module.

F

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.

file header: A portion of a COFF object file that contains general information about the object file (such as the number of section headers, the type of system the object file can be downloaded to, the number of symbols in the symbol table, and the symbol table's starting address).

G

global: A kind of symbol that is either 1) defined in the current module and accessed in another, or 2) 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).

H

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 actual code or data.

I

incremental linking: The linking of files that have already been linked.

initialized section: A COFF section that contains executable code or initialized data. An initialized section can be built up with the .data, .text, or .sect directive.

input section: A section from an object file that will be linked into an executable module.

L

label: A symbol that begins in column 1 of a source statement and corresponds to the address of that statement.

line number entry: An entry in a COFF output module that maps lines of assembly code back to the original C source file that created them.

linker: A software tool 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 SPC.

loader: A device that loads an executable module into system memory.

M

member: The elements or variables of a structure, union, or enumeration.

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 source statements that are substituted for the macro call and are subsequently assembled.

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.

magic number: A COFF file header entry that identifies an object file as a module that can be executed by the MSP430.

map file: An output file, created by the linker, that shows the memory configuration, section composition, and section allocation, as well as symbols and the addresses at which they were defined.

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.

N

named section: An initialized section that is defined with a .sect directive, or an uninitialized section that is defined with a .usect directive.

O

object file: A file that has been assembled or linked and contains machine–language object code.

object format converter: A program that converts COFF object files into Intel–format, Tektronix–format, TI–tagged format, or Motorola–S format object files.

object library: An archive library made up of individual object files.

operand: The arguments, or parameters, of an assembly language instruction, assembler directive, or macro directive.

optional header: A portion of a COFF object file that the linker uses to perform relocation at download time.

options: Command parameters that allow you to request additional or specific functions when you invoke a software tool.

output module: A linked, executable object file that can be downloaded and executed on a target system.

P

partial linking: The linking of a file that will be linked again.

R

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.

S

section: A relocatable block of code or data that will ultimately occupy contiguous space in the memory map.

section header: A portion of a COFF object file that contains information about a section in the file. Each section has its own header; the header points to the section's starting address, contains the section's size, etc.

section program counter: See SPC.

sign–extend: To fill the unused MSBs of a value with the value's sign bit.

SPC (section program counter): An element of the assembler that keeps track of the current location within a section; each section has its own SPC.

static: A kind of 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 re–entered.

storage class: Any entry in the symbol table that indicates how a symbol should be accessed.

string table: A table that stores symbol names that are longer than 8 characters (symbol names of 8 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.

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 so that it can be used by a debugging tool such as a simulator or an emulator.

symbol table: A portion of a COFF object file that contains information about the symbols that are defined and used by the file.

T

tag: An optional “type" name that can be assigned to a structure, union, or enumeration.

target memory: Physical memory in a MSP430–based system into which executable object code is loaded.

.text: One of the default COFF sections; an initialized section that contains executable code. You can use the .text directive to assemble code into the .text section.

U

unconfigured memory: Memory that is not defined as part of the memory map and cannot be loaded with code or data.

uninitialized section: A COFF section that reserves space in the memory map but that has no actual contents. These sections are built up with the .bss, .reg, and .usect directives.

union: A variable that may hold (at different times) objects of different types and sizes.

unsigned: A kind of value that is treated as a positive number, regardless of its actual sign.

W

well–defined expression: An expression 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.

Topics

26 Floating Point Formats

All MSP430 floating-point formats consist of three fields: an exponent field (e), a single-bit sign field (s), and a fraction field (f). The sign field and fraction field may be considered as one unit and referred to as the mantissa field. The fraction contains an implied most-significant bit, which is always 1 for a correctly represented floating-point constant. This provides an additional bit of precision. The exponent is bias 128; that is, subtract 128 from the unsigned value of the 8 exponent bits to arrive at an actual value for the exponent. A sign, exponent and fraction of zero is used as a special representation of value zero.

26.1 Single Precision Format

In the single precision format, the floating-point number is represented by an 8-bit exponent, a sign bit and a 23-bit fraction.

The format is as follows:

```
e e e e e e e e   s f f f f f f f   f f f f f f f f   f f f f f f f f
```

The fraction contains 23 actual bits plus an implied bit f_0, always representing a 1. The value of each f; is arrived at through this formula:

$$f_i = \frac{1}{2^i} \quad => \quad f = \sum_{i=0}^{23} f_i = \sum_{i=0}^{23} \frac{1}{2^i}$$

Therefore, the layout in terms of values is

$$1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{64}, \frac{1}{128}, \frac{1}{256}, \frac{1}{512}, \ldots$$

Example: Calculating the fraction (80 in those examples is the exponent)

Floating Point Value	f_i	**Fraction Decimal Equivalent**
80100000	$1, \frac{1}{8}$	$1 + \frac{1}{8} = 1.125$
80310000	$1, \frac{1}{4}, \frac{1}{8}, \frac{1}{128}$	$1 + \frac{1}{4} + \frac{1}{8} + \frac{1}{128} = 1.3828125$

Given the above format, some examples of acceptable floating-point values are shown in the following examples.

Example: Calculating Floating-Point Values

81d00000

```
    Exponent       Sign              Fraction
1 0 0 0 0 0 0 1   1 1 0 0 0 0 0 0   0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0
```

The encoded exponent equals 129; the real exponent equals 129-128 = 1.

The fraction equals 1 (implied f_0) $+ \frac{1}{2} + \frac{1}{8} = 1.625$.

The following formula expresses the actual value of the floating-point number:

$$s \times f \times 2^{e-128}$$

where s is the sign of the number (either 1 or -1), f is the value of the fraction ($1.0 \le f < 2.0$) and e is the represented value of the exponent.

Therefore, the floating-point value is

$$-1 \times 1.625 \times 2^{129-128} = -3.25$$

The following list gives other examples of proper floating-point values derived from the above formulas.

80000000h	=>	1.0	81500000h	=>	3.25
80800000h	=>	- 1.0	8f3b8000h	=>	4.8e4
00000000h	=>	0.0	840c0000h	=>	1.75e1
83200000h	=>	1.0e1	79937500h	=>	- 9.0e-3

26.2 Double Precision Format

The only difference to the single precision format is the length of the fraction:

```
|e e e e e e e e| |s f f f f f f f| |f f f f f f f f| |f f f f f f f f| |f f f f f f f f| |f f f f f f f f|
```

Here it contains 39 actual bits plus an implied bit fo; so the summation formation for the fraction changes to:

$$f = \sum_{i=0}^{39} f_i = \sum_{i=0}^{39} \frac{1}{2^i}$$

Index

Notes

TI SC Sales Offices in Europe

Belgium
Texas Instruments S.A./N.V.
Brussels
Tel.: (02) 7 26 75 80
Fax: (02) 7 26 72 76

Finland
Texas Instruments OY
Espoo
Tel.: (0) 43 54 20 33
Fax: (0) 46 73 23

France, Middle-East & Africa
Texas Instruments
Velizy Villacoublay
Tel.: (1) 30 70 10 01
Fax: (1) 30 70 10 54

Germany
Texas Instruments GmbH
Freising
Tel.: (0 81 61) 80-0
Fax: (0 81 61) 80 45 16

Hannover
Tel.: (05 11) 90 49 60
Fax: (05 11) 6 49 03 31

Ostfildern
Tel.: (07 11) 3 40 30
Fax: (07 11) 3 40 32 57

Holland
Texas Instruments B.V.
Amstelveen
Tel.: (0 20) 6 40 04 16
Fax: (0 20) 5 45 06 60
(0 20) 6 40 38 46

Hungary
TI Representation:
Budapest
Tel.: (1) 1 76 37 33
Fax: (1) 2 02 62 56

Italy
Texas Instruments S.p.A.
Agrate Brianza (Mi)
Tel.: (0 39) 6 84 21
Fax: (0 39) 6 84 29 12

Republic of Ireland
Texas Instruments Ltd.
Dublin
Tel.: (01) 4 75 52 33
Fax: (01) 4 78 14 63

Spain
Texas Instruments S.A.
Madrid
Tel.: (1) 3 72 80 51
Fax: (1) 3 72 82 66

Sweden
Texas Instruments
International Trade Corporation
Kista
Tel.: (08) 7 52 58 00
Fax: (08) 7 51 97 15

United Kingdom
Texas Instruments Ltd.
Northampton
Tel.: (0 16 04) 66 30 00
Fax: (0 16 04) 66 30 01

TEXAS INSTRUMENTS

TI Technology Centres

France
Texas Instruments
Velizy Villacoublay
Tel.: Standard:
(1) 30 70 10 01
Technical Service:
(1) 30 70 11 33

Holland
Texas Instruments B.V.
Amstelveen
Tel.: (0 20) 5 45 06 00
Fax: (0 20) 6 40 38 46

Italy
Texas Instruments S.p.A.
Agrate Brianza (Mi)
Tel.: (0 39) 6 84 21
Fax: (0 39) 6 84 29 12

Sweden
Texas Instruments
International Trade Corporation
Kista
Tel.: (08) 7 52 58 00
Fax: (08) 7 51 97 15

European SC Information Centre

Telephone:

Dutch	(33) 1 30 70 11 66
English	(33) 1 30 70 11 65
French	(33) 1 30 70 11 64
German	(33) 1 30 70 11 68
Italian	(33) 1 30 70 11 67
Fax:	(33) 1 30 70 10 32

Printed in Germany
by Sellier Druck, Angerstr. 54, 85354, Freising

Chlorine free paper – to protect our environment

TEXAS INSTRUMENTS

SLAUE12