TEXAS INSTRUMENTS

TMS370 Family Assembly Language Tools

User's Guide

1987

8-Bit Microcontroller Family

TMS370 Family Assembly Language Tools User's Guide

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes in the devices or the device specifications identified in this publication without notice. TI advises its customers to obtain the latest version of device specifications to verify, before placing orders, that the information being relied upon by the customer is current.

In the absence of written agreement to the contrary, TI assumes no liability for TI applications assistance, customer's product design, or infringement of patents or copyrights of third parties by or arising from use of semiconductor devices described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor devices might be or are used.

Contents

Illustrations

Tables

1. Introduction

The TMS370 devices are well supported by a full set of hardware and software development tools. This document discusses the software development tools that are included with the TMS370 assembly language package:

- Assembler
- Archiver
- Linker
- Code Conversion Utility

These tools can be installed on the either of the following PC systems:

- IBM-PC with PC-DOS
- TI-PC with MS-DOS

The TMS370 assembly language tools create and use object files that are in common object file format, or **COFF**. COFF object format encourages and facilitates modular programming. Object files contain separate blocks (called *sections*) of code and data that you can load into different TMS370 memory spaces. You will be able to program the TMS370 more efficiently if you have a basic understanding of COFF; Section 3, Introduction to Common Object File Format, discusses this object format in detail.

Topics covered in this introductory section include:

1.1 Software Development Tools Overview

Figure 1-1 shows the TMS370 assembly language development flow. The center section of the illustration highlights the most common path; the other portions are optional.

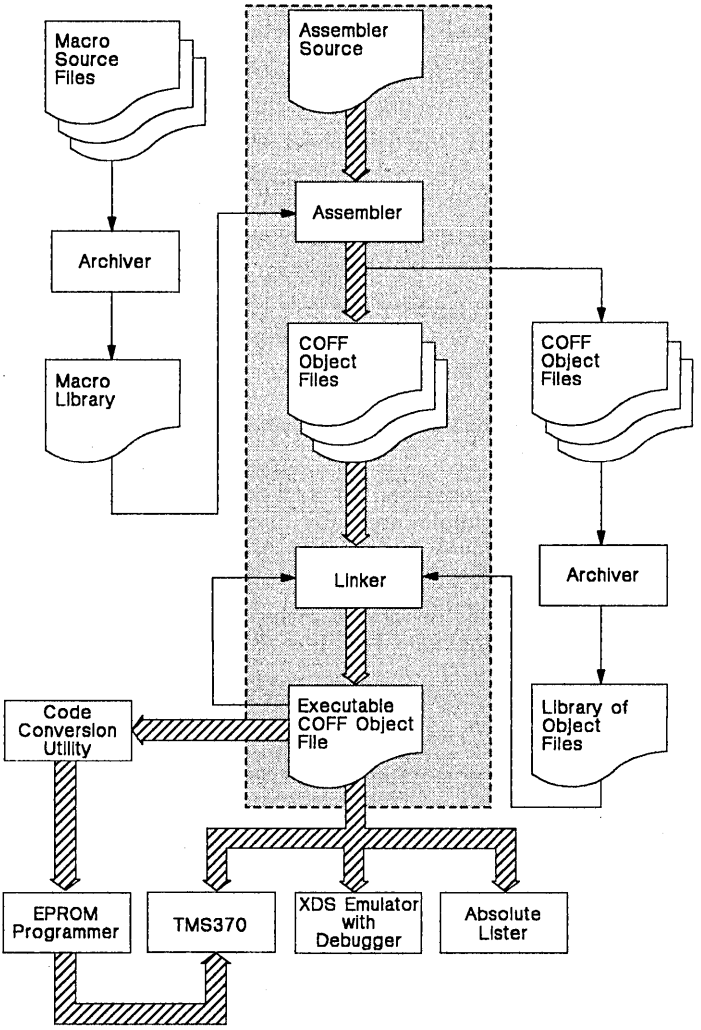


Figure 1-1. TMS370 Assembly Language Development Flow

- The **assembler** translates assembly language source files into machine language object files. Source files can contain instructions (discussed in the *TMS370 Family User's Guide*), assembler directives (discussed in Section 4), and macro directives (discussed in Section 7). Assembler directives control various aspects of the assembly process, such as the source listing format, symbol definition, and how the source code is placed into sections.

- The **archiver** allows you to collect a group of files into a single archive library. For example, you could collect several macros together into a macro library. The assembler can search through the library and use the members that are called as macros by a source file. You can also use the archiver to collect a group of object files into an object library. The linker will include the library 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 can also accept archive 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 specific addresses or within specific portions of TMS370 memory, and define or redefine global symbols.

- The main purpose of this development process is to produce a module that can be executed in a system that contains a **TMS370 device**. You can use one of several debugging tools to refine and correct your code before downloading it to a TMS370 system.

  - The **XDS/22 emulator with symbolic debugger** is a realtime, in-circuit emulator with a screen-oriented interface. It provides symbolic debugging. The debugger is not shipped as part of the TMS370 Assembly Language Package.

  - The **absolute lister** provides a listing of the absolute addresses of an object file.

- Most EPROM programmers do not accept COFF object files as input. The **code conversion utility** converts a COFF object file into Intel hex object format. The converted file can be downloaded to an EPROM programmer.

1.2 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 entire user's guide. These examples show the most common methods for invoking the assembler and linker.

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

file1.asm

```
        .global incqw
start   clr     r20
        clr     r21
        clr     r22
        clr     r23
loop    call    incqw   ;loop
        jnc     loop
        .end
```

file2.asm

```
        .global incqw
incqw   incw    #1,r23
        jnc     skp
        incw    #1,r21
skp     rts
        .end
```

2) Assemble `file1.asm`. Enter: **`asm370 file1`**

   This example creates an object file called `file1.obj`. The assembler always creates an object file. You can specify a name for the object file, but if you don't, the assembler will use the input filename appended to the .obj extension. Notice that you didn't specify an extension for `file1`. The assembler assumes that the input file has an extension of **.asm**.

   Now assemble `file2.asm`. Enter: **`asm370 file2 -l`**

   This time, the assembler creates an object file called `file2.obj`. The -l option told the assembler to create a listing file; the listing file for this example is called `file2.lst`.

3) Link `file1.obj` and `file2.obj`. Enter: **`lnk370 file1 file2`**

   The linker assumes that `file1` and `file2` have an extension of **.obj**. The linker combines these two files to create an executable object module with a default name of `a.out`.

You can find more information about invoking the tools in the following sections:

1.3 Manual Organization

This document contains the following sections; it also contains several appendices, an index, and a reference card.

Section 1 Introduction
This section provides an overview of the assembly language tools and the assembly language development process, provides quick examples for invoking the assembly language tools, lists related documentation, and explains the style and symbol conventions used throughout the document.

Section 2 Software Installation
This section contains instructions for installing the assembly language tools on IBM-PC/PC-DOS and TI-PC/MS-DOS and systems.

Section 3 Introduction to Common Object File Format
Read Section 3 before using the assembler and linker. Common object file format, or COFF, is the object file format that the TMS370 assembly language tools use. This section discusses the basic COFF concept of **sections** and how they can help you to use the assembler and linker more efficiently. (Appendix A contains specific information about COFF object file structure; its main purpose is to provide you with information about symbolic debugging.)

Section 4 Assembler Description
This section tells you how to invoke the assembler, and then discusses source statement format, valid constants and expressions, and assembler output.

Section 5 Assembler Directives
This section is divided into two parts. The directives can be easily categorized by function, and the first part of this section describes the directives according to function. The second part of this section is a reference that presents the directives in alphabetical order.

Section 6 Instruction Set Summary
This section summarizes the TMS370 instruction set.

Section 7 Macro Language
This section tells you how to create and use macros.

Section 8 Archiver Description
This section tells you how to invoke the archiver to create archive libraries.

Section 9 Linker Description
This section tells you how to invoke the linker, provides details of linker operation, discusses linker directives, and presents a detailed linking example.

Section 10 Absolute Lister Description
This section tells you how to invoke the absolute lister so that you can obtain a listing of the absolute addresses of an object file.

Section 11 Code Conversion Utility Description
This section tells you how to invoke the code conversion utility so that you can convert a COFF object file into an Intel hex object file format.

1.4 Related Documentation

The following TMS370 documents are also available.

- The ***TMS370 Family User's Guide*** discusses hardware aspects of the TMS370, such as pin functions, architecture, stack operation, and interfaces, and contains the TMS370 instruction set. (If you received this User's Guide with the TMS370 Assembly Language Tools package, you should also have received a copy of the *TMS370 User's Guide*).

- The ***TMS370C050/TMS370C850 Data Sheet*** and the ***TMS370C010/TMS370C810 Data Sheet*** contain the recommended operating conditions, electrical specifications, and timing characteristics for the following devices:

  - TMS370C050
  - TMS370C850
  - TMS370C010
  - TMS370C0810

- The ***TMS370 Family EEPROM Programmer User's Guide*** describes the installation and operation of the EEPROM programmer,

- The ***TMS370 Family XDS Debugger User's Guide*** describes installation and operation of the TMS370 XDS/22 emulator. The XDS emulator provides symbolic debugging and a screen-oriented interface.

1.5 Style and Symbol Conventions

In this user's guide, interactive displays and programming examples are shown in a `special font`. Table 1-1 contains other style and symbol conventions that are used throughout this document.

Table 1-1. Symbol and Abbreviation Definitions

Symbol	Definition	Symbol	Definition
R0–R255	Extended registers 0 through 255	P0–P255	Peripheral registers 0 through 255
A	Accumulator A	B	Accumulator B
PC	Program counter register	SP	Stack pointer register
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 enclosed in brackets with an appropriate substitute. For example, substitute an actual label for <label>; substitute an actual destination expression for <expression>.		

2. Software Installation

This section contains step-by-step instructions for installing the TMS370 assembly language tools package. This package can be installed on the IBM PC (running PC-DOS[1]) and the TI PC (running MS-DOS)[2]. Section 1.5 (page 1-7) lists style and symbol conventions that are used in this section.

The TMS370 software package is shipped on one double-sided, dual-density diskette. The tools execute in batch mode. At least 512K bytes of memory space must be available in your system.

These instructions are for both hard disk systems and dual floppy drive systems. On a dual-drive system, the PC/MS-DOS system diskette should be in drive B. The instructions use these symbols for drive names:

A: Floppy disk drive for hard disk systems *or* source drive for dual-drive systems.
B: Destination or system disk drive for dual-drive systems.
C: Winchester (hard disk) for hard disk systems. (**E:** on TI PCs.)

1) Make backups of the product diskettes. First format a blank diskette. Insert a blank (destination) diskette in drive A. Enter:

```
FORMAT A: <CR>
```

When PC/MS-DOS prompts: `FORMAT ANOTHER (Y/N)?`, respond with `N`. Now copy the disks.

On *hard disk* systems, enter:

```
DISKCOPY A:  A: <CR>
```

Follow the system prompts, removing and inserting the product and blank diskettes as directed. When PC/MS-DOS prompts: `COPY ANOTHER (Y/N)?`, respond with `N`.

On *dual-drive* systems, place a product diskette in drive A: and a blank, formatted diskette in drive A. Enter:

```
COPY A:*.*  B:*.* <CR>
```

2) Create a directory to contain the TMS370 software.

On *hard disk* systems, enter: `MD C:\370TOOLS <CR>`

On *dual-drive* systems, enter: `MD B:\370TOOLS <CR>`

3) Copy the TMS370 tools onto the hard disk or the system disk.

On *hard disk* systems, enter: `COPY A:\*.* C:\370TOOLS\*.* <CR>`

On *dual-drive* systems, enter: `COPY A:\*.* B:\370TOOLS\*.* <CR>`

---

1 PC-DOS is a trademark of International Business Machines.

2 MS is a trademark of Microsoft Corporation.

3. Introduction to Common Object File Format

The assembler and linker create object files that can be executed by a TMS370 device. The format that these object files are in is called *common object file format*, or ***COFF***.

COFF object format encourages and facilitates modular programming. When you write a TMS370 assembly language program, you should think in terms of *blocks* of code and data. These blocks are known as **sections**.

This chapter provides an overview of COFF sections and includes the following topics:

Appendix A contains more advanced information about COFF. It discusses details about the actual object file structure, such as the fields in a file header and the structure of a symbol table entry. Appendix A is mainly useful for those of you who are interested in symbolic debugging.

3.1 Sections

The smallest relocatable unit of an object file is called a **section**. A section is a relocatable block of code or data which will (ultimately) occupy contiguous space in the TMS370 memory map. Each section of an object file is separate and distinct from the other sections. COFF object files always contain four default sections:

.reg Contains uninitialized space in the register file
.bss Contains uninitialized data
.data Contains initialized data variables
.text Contains executable code

In addition, the assembler and linker allow you to create, name, and link **named** sections that can be used similarly to the .data and .text sections.

It is important to note that there are two basic types of sections:

- **Initialized sections** contain data or code; the .text, .data, and named sections are initialized.
- **Uninitialized sections** reserve space in the memory map for uninitialized data; the .bss and .reg sections are 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 similarly to the object file shown in Figure 3-1.

One of the linker's functions is to place sections into the target memory map (this is called **allocation**). Since 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 the target memory map. For example, you could define a section that contains an initialization routine, and then use the linker to allocate the routine to the portion of the memory map that contains ROM.

Figure 3-1 shows the relationship between sections in an object file and a hypothetical target memory.

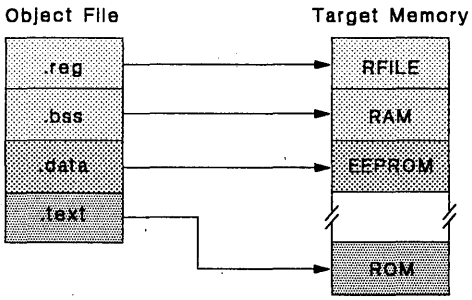


Figure 3-1. Partitioning Memory into Logical Blocks

3.2 How the Assembler Handles Sections

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

- The **.bss** directive reserves a defined amount of space in uninitialized RAM that can be used for storing data.
- The **.reg** and **.regpair** directives reserve a block of memory for relocatable registers in the register file.
- The **.text** directive identifies the source statements that follows it as executable code. The statements following a .text directive are assembled into the .text default section.
- The **.data** directive identifies the source statements that follows it as initializable data. The statements following a .data directive are assembled into the .data default section.
- The **.sect** directive defines a named section, and identifies the source statements that follow it as belonging to that named section. The statements following a .sect directive are assembled into the appropriate named section.

The .bss, .reg, and .regpair directives create *uninitialized sections*; the .text, .data, and .sect directives create *initialized sections*. Section 3.2.1 and Section 3.2.2 discuss some of the details about these directives. Section 3.2.4 shows an example of using the sections directives.

> **Note:**
>
> If you don't use any of the sections directives, the assembler will assemble all code into the .text section.

3.2.1 Uninitialized Sections

Uninitialized sections reserve space in the TMS370 memory map for creating and storing uninitialized variables. These sections have no contents; they simply reserve memory. Although these sections do not increase the size of the object file, they do increase the amount of memory needed to run a program.

The TMS370 assembly language tools support two types of uninitialized sections:

- The **.bss** section, which is built by using the .bss directive. The .bss section reserves memory space for uninitialized data. It is referenced by a *16-bit address*, because the .bss section can be relocated anywhere in the address space.
- The **.reg** section, which is built by using the .reg or .regpair directives. The .reg section reserves memory space for relocatable registers. It is referenced by an *8-bit address*, and is *always* relocated in the first 256 bytes of memory (the register file).

The final .bss or .reg section that the assembler creates may consist of any number of individually allocated data areas.

The syntax for these directives is:

```
.bss     <symbol>,<size>
.reg     <symbol>,<size>
.regpair <symbol>,<size>
```

The `symbol` points to the first *byte* reserved by the .bss or .reg directive; it points to the *last* byte reserved by the .regpair directive. The symbol can be referenced by any other section and can also be declared external (by the .global or .globreg assembler directives). A symbol that is declared with a .reg or .regpair directive is called a *relocatable-register symbol*; you can use it like any other register symbol (r1, r2, etc.), but the linker will determine its address within the register file.

The `size` parameter is an absolute expression. The .bss directive reserves *size* **bytes** in the .bss section; the .reg and .regpair directive reserve *size* **bytes** in the .reg section.

3.2.2 Initialized Sections

Initialized sections contain executable code or initialized data. The contents of these sections are stored in the object file and placed in the TMS370 memory map when the program is loaded. Each initialized section is separately relocatable and may contain symbolic references to objects defined in any other section. The linker automatically resolves these section-relative references.

There are three directives that tell the assembler to place code or data into a section: .text., .data, and .sect. When the assembler encounters one of these directives, it will stop assembling into the current section (acting as an implied "end current section" instruction). It will then assemble subsequent code into the respective section until it encounters another .text, .data, or .sect directive.

The .sect directive allows you to create a new, *named* section that can be used like the default .text and .data sections. You can use the .sect directive to create up to 32,767 separate sections. (The name can only be 8 characters long.)

3.2.3 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 and data, it increments the appropriate SPC. If you *resume* assembling code into a section, the assembler will remember the appropriate SPC's previous value and continue incrementing at that point.

When assembled, each section begins at address 0; the linker will relocate sections according to their locations in the memory map.

3.2.4 An Example That Uses Sections Directives

Figure 3-2 (page 3-6) contains an example that shows how you can build COFF sections incrementally, swapping back and forth between the different sections. You can use sections directives:

- To begin assembling code into a section for the first time,

  or

- To 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 format of this example is a list file. By using a list file, this example shows how these counters are modified during assembly. A line in a list file has four fields:

1) The first field contains the SPC value.

2) The second field contains the object code.

3) The third field contains the line counter.

4) The fourth field contains the original source statement.

Note that a .bss, .reg, or .regpair directive can appear anywhere in an initialized section without affecting the contents of the initialized section. Initialized section directives end the current section and begin a new section. The .bss, .reg, and .regpair directives **do not** end the current section and begin a new one; they simply "escape" from the current section temporarily.

```
0000                0001 ;**********************************************
0000                0002 ;*   Start assembling into the .text section   *
0000                0003 ;**********************************************
0000                0004         .text
0000 0102           0005         .byte  1,2
0002 0304           0006         .byte  3,4
0004                0007
0004                0008 ;**********************************************
0004                0009 ;*   Start assembling into the .data section   *
0004                0010 ;**********************************************
0004                0011         .data
0000 070809         0012         .byte  7,8,9
0003                0013
0003                0014 ;**********************************************
0003                0015 ;*  Define a section named var_defs and begin  *
0003                0016 ;*  assembling code into it                    *
0003                0017 ;**********************************************
0003                0018         .sect  "var_defs"
0000 0B0C           0019         .byte  11,12
0002                0020
0002                0021 ;**********************************************
0002                0022 ;*  Continue assembling into .text             *
0002                0023 ;**********************************************
0002                0024         .text
0004 05             0025         .byte  5
0000                0026         .bss   sym,12      ; Reserve 12 bytes in .bss
0005 06             0027         .byte  6           ; Still in .text
0006                0028
0006                0029 ;**********************************************
0006                0030 ;*  Continue assembling into .data             *
0006                0031 ;**********************************************
0006                0032         .data
0003 0A             0033         .byte  10
0000                0034         .reg   RR1,8       ; Reserve 8 bytes in .reg
0004 0B             0035         .byte  11          ; Still in .data
0005                0036
0005                0037 ;**********************************************
0005                0038 ;*  Continue assembling into var_defs          *
0005                0039 ;**********************************************
0005                0040         .sect  "var_defs"
0002 0D0E           0041         .byte  13,14
```

Figure 3-2. Using Sections Directives

After assembly, the sections will contain the following:

.text	Contains bytes 1, 2, 3, 4, 5, and 6
.data	Contains bytes 7, 8, 9, 10, and 11
var_defs	Contains bytes 11, 12, 13, and 14
.bss	Reserves 10 bytes for uninitialized data
.reg	Reserves 8 bytes for relocatable register data

3.3 How the Linker Handles Sections

The linker uses the sections in COFF object files as building blocks to create output sections in an executable COFF output module. The linker can then place the output sections into specified portions of the target memory. Usually, the linker simply combines all input sections that have the same name into one output section that has this same name. (For example, the linker would combine the two .text sections from two input files to create one .text section in the executable object module.)

3.3.1 Default Memory Allocation

Figure 3-3 shows a simple example of how two files might be linked together.

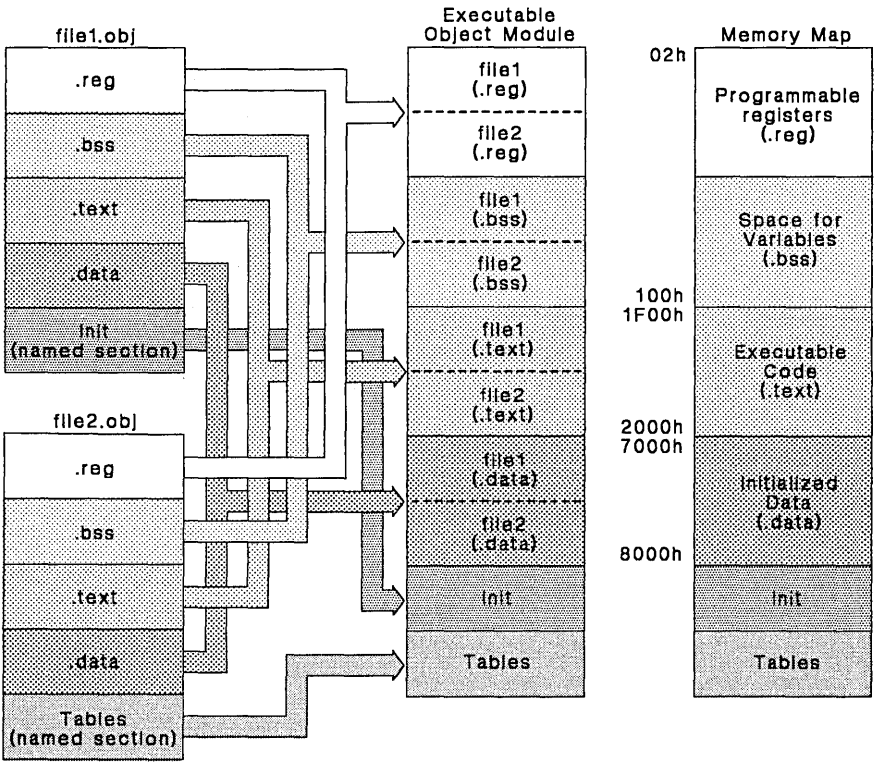


Figure 3-3. Combining Input Sections to Form an Executable Object Module

In Figure 3-3, `file1.obj` and `file2.obj` are files that have been assembled and are going to be used as linker input. They each contain 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; by default, the linker will begin at address 02h (after registers A and B) and place the sections one after the other as shown.

3.3.2 Placing Sections in the Memory Map

Figure 3-3 (page 3-7) 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 are comprised of 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 linker provides two directives that support these functions:

- 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.

4. Assembler Description

The TMS370 assembler translates assembly language source files into machine language object files. These object files are in common object file format (COFF), discussed in Section 3. Figure 4-1 (page 4-2) illustrates the assembler's role in the overall assembly language development flow. Source files can contain these assembly language elements:

- Assembly language instructions (summarized in Section 6),
- Assembler directives (described in Section 5), **and**
- Macro directives (described in Section 7).

The assembler is a one-pass assembler. It performs the following operations as it processes the source statements in a text file to produce an object code file and an optional listing file:

- Reads the source file.
- Calculates the values of labels and symbols.
- Builds the symbol table.
- Maintains a *section program counter* (**SPC**) for each section of object code generated. The SPC defines the virtual program memory addresses assigned to the associated object code. The assembler uses the SPC while it builds the symbol table.
- Prints error messages for invalid source lines.
- Generates the object file.
- Generates an optional listing file and an optional cross-reference file (if requested).

Topics in this section include:

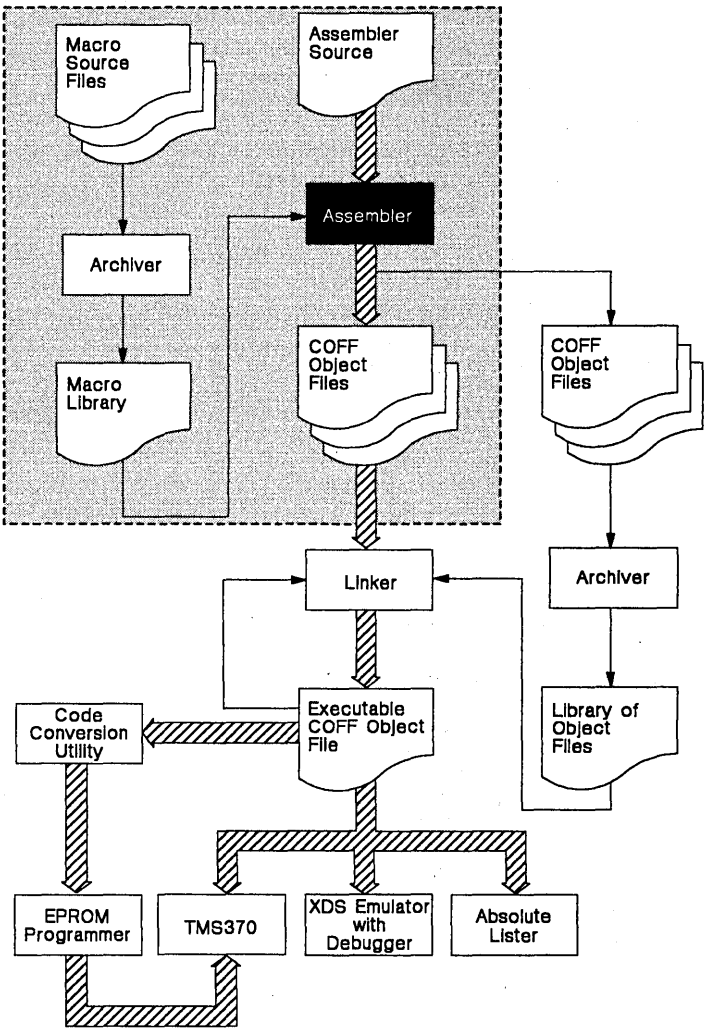


Figure 4-1. Assembler Development Flow

4.1 Invoking the Assembler

To invoke the assembler, enter:

```
asm370  <input file>  [<object file> [<listing file>]][-<options>]
```

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

listing file is the name of the listing file that the assembler creates. If you do not supply an extension, the assembler will use ***.lst*** as a default extension. If you do not supply a name for a listing file, *the assembler will not create one*, unless you use the -l option. In this case, the assembler will use the input file name with the *.lst* extension.

options indicate which assembler options you are using. Case is insignificant for assembler options. Options may appear anywhere on the command line; precede each option with a hyphen (-). You can string the options together; for example, **-lx** is equivalent to **-l -x**. Valid assembler options include:

- **-l** (lowercase "L") Produce a listing file. If you do not specify a listing file name, the assembler will create one; the default filename is the input file name with an extension of .lst.
- **-x** Produce a cross-reference table and append it to the end of the listing file. If you did not request a listing file, the assembler will create one anyway, but the listing will only contain the cross-reference table.
- **-q** Quiet run. The assembler normally prints error messages on the screen; if you specify the -q option, the assembler will put the error messages in a disk file.
- **-a** Create an absolute listing. When you use -a, the assembler does not produce an object file. This absolute listing file is used in conjunction with the absolute lister.

You may want to create a batch file for assembling and linking. The assembler issues an exit code that equals the number of errors that occur during an assembly. In a batch file, you an use the MS/PC-DOS *IF ERRORLEVEL* command to see if the exit code is greater than or equal to the specified number. If there are more than a specified number of assembly errors, (1 in the following example), the file won't be linked. Here is a sample batch file that you might want to use:

```
asm370 %1 -l
IF ERRORLEVEL 1 GOTO ERR
lnk370 %1 -o %1.out -m %1.map
:ERR
```

4.2 Source Statement Format

TMS370 assembly language source programs consist of source statements that may contain assembler directives, assembly language instructions, macro directives, and comments. The assembler accepts the ASCII character set (see Appendix D). Source statement lines may be as long as the source file format allows. The assembler reads up to 256 characters per line; any characters beyond 256 are ignored.

The next several lines show examples of source statements:

```
SYM     .equ   0A5h          ; Symbol SYM = 0A5h
Begin:  ADD    #SYM+5,R1     ; Add (SYM+5) to the contents of R1
        MOV    R1,R2         ; Move contents of R1 to R2
```

A source statement may contain four ordered fields. The general syntax for source statements is:

```
[<label>[:]]  <mnemonic>  [<operand list>]  [;<comment>]
```

where:

- Labels are optional; if used, they must begin in column 1.
- Comments are optional; if used, they must begin with a semicolon (;).
- One or more blanks must separate each field.
- Statements must begin with a label, a blank, or a comment indicator.

4.2.1 Label Field

Labels are optional for all assembly language instructions and for most (but not all) assembler directives. 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 $) and may be followed by a colon (:). Labels are case sensitive, and the first character cannot be a number. If you don't use a label, then the first character position must contain a blank or a comment indicator.

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 102h.

```
0102 00AA016D64    0007 Start:      .word    0AAh,365,"words"
```

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:   .equ    $     ; $ represents 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):

```
0124               0011 Init:
0124 000A          0012             .word    10
```

4.2.2 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, .equ, .list)
- Macro directive (such as .ASG, .MACRO, .VAR)
- A macro mnemonic (macro call)

4.2.3 Operand Field

The operand field is a list of operands that follows the mnemonic field. An operand can be a constant (see Section 4.3), a symbol (see Section 4.5), or a combination of constants and symbols in an expression (see Section 4.6). You must separate operands with commas.

4.2.3.1 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 - operand is an address.** If you do not use a prefix with an operand, the assembler will treat the operand as an address. Here is an example of an instruction that uses operands without prefixes:

  ```
  ADD     r123,B
  ```

  The operands *r123* and *B* are addresses (in this case, they're the addresses of registers). The assembler will add the *contents* of r123 to the *contents* of register B.

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

  ```
  ADD     #123,B
  ```

  The operand *#123* is an immediate value. The assembler will add 123 to the contents of register B.

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

  ```
  MOV     @R4,A
  ```

  The operand *@R4* specifies an indirect address. The assembler will go to the *address* specified by the contents of register pair R3:R4, and then move the contents of that location to register A.

4.2.3.2 Register Aliasing

Many instructions that use registers as operands have short forms that use registers A and B. When you use register A or B as an operand for one of these instructions, the assembler will use a different opcode and generate a shorter instruction.

Register A is equivalent to register r0, and register B is equivalent to register r1. If you specify r0 or r1, and the instruction allows you to use A or B instead, the assembler will optimize the instruction and use the shorter form. If you specify A or B, and the instruction allows r*n* but does not allow A or B, the assembler will correct the instruction for you. These are the optimizing/correcting rules that the assembler follows:

- If **A** is illegal, the assembler will use **r0**
- If **B** is illegal, the assembler will use **r1**
- If **r0** is illegal or not optimal, the assembler will use **A**
- If **r1** is illegal or not optimal, the assembler will use **B**

Here are some examples:

Code	Opcode Generated	Equivalent Code
`CMP r09,r1`	`3D 09`	`CMP r9,A`
`CMP r012,A`	`1D 12`	`CMP r12,A`
`CMP A,r011`	`4D 00 11`	`CMP r0,r11`

In the first example, register r1 is specified as an operand. The assembler will optimize this instruction by substituting register B for register r1, which allows it to use the shorter opcode for the `CMP rn1,B` form. In the second example, register A is specified and used correctly. The third example, however, uses register A incorrectly; in this case, the assembler substitutes r0 for A to correct the error.

4.2.3.3 Immediate Addressing for Directives

The immediate addressing mode is generally most useful when used 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, A
.byte   10
```

In the first statement, immediate addressing mode is necessary to tell the assembler to add the *value* 10 to register A. If immediate addressing mode had not been used, the assembler would have treated 10 as an address, and added the *contents* of location 10 to register A. 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.

In some cases, you can use immediate addressing for directive operands. Here is a legal example:

```
.byte   #R1
```

R1 is an address (1); therefore, `.byte R1` is not a legal statement. Using immediate addressing in this statement causes the assembler to use the address of R1 as a value.

4.2.4 Comment Field

A comment **must** begin with a semicolon (;).

A comment can begin in any column, and extends to the end of the source line. A source statement that contains only a comment is valid. A comment can contain any ASCII character, including blanks. Its contents are printed in the assembly source listing but they do not affect the assembly.

4.2.5 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 only be declared 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 **only** be used as an operand for an 8-bit jump instruction.

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

1) By the .newblock directive
2) By changing sections (using a .sect, .text, or .data directive)
3) By entering or leaving an include file (specified by the .include directive)

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

```
Label1:   mov    r2,r3
          jnz    $1
          mov    #-1,r3
$1        cmp    r3,A
          .newblock      ; Undefine $1 so it can be used again
          jne    $1
          inc    r3
$1        add    r3,r4
```

The following code uses a local label **illegally**:

```
Label1:   mov    r2,r3
          jnz    $1
          mov    #-1,r3
$1        cmp    r3,A
          jne    $1
          inc    r3
$1        add    r3,r4    ; 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 will issue a multiple-definition error. However, if you use a local label within a macro and then use .newblock within the macro, the local label will be 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.

4.3 Constants

The assembler supports six types of constants:

- Binary integer constants,
- Octal integer constants,
- Decimal integer constants,
- Hexadecimal integer constants,
- Character constants, **and**
- 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 0FFFFH is equal to $0000FFFF_{16}$ or $65{,}535_{10}$; it **does not** equal -1.

4.3.1 Binary Integers

A binary integer constant is a string of up to 32 binary digits (0s and 1s) followed by the suffix **B** (or **b**). If less than 32 digits are specified, the assembler will right justify the bits. Examples of valid binary constants include:

00000000B Constant equal to 0

0100000b Constant equal to 32_{10}

01b Constant equal to 1_{10}

11111000B Constant equal to 248_{10}

4.3.2 Octal Integers

An octal integer constant is a string of octal digits (0 through 7) followed by the suffix **Q** (or **q**). Examples of valid octal constants include:

10Q Constant equal to 8_{10}

100000Q Constant equal to $32{,}768_{10}$

226Q Constant equal to 150_{10}

4.3.3 Decimal Integers

A decimal integer constant is a string of decimal digits, ranging from -2,147,483,647 to 4,294,967,295. Examples of valid decimal constants include:

1000 Constant equal to 1000_{10} or $3E8_{16}$

-32768 Constant equal to $-32{,}768_{10}$ or $FFFF8000_{16}$

25 Constant equal to 25_{10} or 19_{16}

4.3.4 Hexadecimal Integers

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

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 $37AC_{16}$

4.3.5 Characters

A character constant is a string of one to four letters enclosed in single or double quotes. The characters are represented internally as 8-bit ASCII characters. A character constant consisting only of two single quotes (no letter) is valid and is assigned the value 0. If less than four characters are specified, the assembler will right justify the bits. Character constants may be used anywhere a numerical constant is used; the assembler converts character constants to numbers. Examples of valid character constants include:

'ab' Represented internally as 00006162_{16}

'C' Represented internally as 00000043_{16}

'abcd' Represented internally as 61626364_{16}

4.3.6 Assembly-Time Constants

If you use the .equ directive to assign a constant value to a symbol, the symbol becomes an assembly-time 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,R0
```

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   R24
       MOV    10,sym
```

4.4 Character Strings

A character string is a string of characters enclosed in single or 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 D lists valid characters.

Examples of valid character strings include:

"sample program"	Defines a 14-character string, `sample program`
"PLAN ""C"""	Defines an 8-character string, `PLAN "C"`

4.5 Symbols

Symbols are used as labels and in operands. A symbol name is a string of alphanumeric characters (A–Z, a–z, 0–9, $, and _). Only the first 32 characters are significant. The first character in a symbol cannot be a number, and a symbol name cannot contain embedded blanks. The symbols you define are case sensitive; for example, the assembler will recognize ABC, Abc, and abc as three unique symbols. (You can override this with the -c assembler option.) User-defined symbols are valid only during the assembly in which they are defined, unless you use the .global directive to declare them as external symbols.

Symbols that are used as **labels** become symbolic addresses that are associated with locations in the program. Labels should be unique names; do not re-use them for other statements. Mnemonic opcodes and assembler directive names (without the . prefix) are valid label names.

Symbols that are used in **operands** must be defined in the assembly by appearing as labels **or** as operands of a .global, .globreg, .equ, .bss, .reg, or .regpair directive. Note that declaring a symbol as external makes it a 16-bit constant.

Symbols can be absolute or relocatable. An absolute symbol's value is a number; a relocatable symbol's value is an address.

The assembler has several predefined symbols, including:

- $ (the dollar sign character) represents the current value of the section program counter (SPC).
- **Port names**, which are of the form P*n* or p*n*, where *n* is an expression that evaluates in the range 0–255 (if the number is greater than 255, the symbol is not considered a register symbol). The number may be decimal or hexadecimal; to specify a hexadecimal number for a register symbol, precede the number with a 0. For example, p010 and p16 are equivalent (the first is hexadecimal, the second is decimal).
- **Register symbols**, which are of the form R*n* or r*n*, where *n* is an expression that evaluates in the range 0–255 (if the number is greater than 255, the symbol is not considered a register symbol). The number may be decimal or hexadecimal; to specify a hexadecimal number for a register symbol, precede the number with a 0. For example, r010 and r16 are equivalent (the first is hexadecimal, the second is decimal). Note that **A** and **B** are valid register symbols; they represent r0 and r1, respectively.

4.6 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 -2,147,483,647 to 4,294,967,295.

Three main factors influence the order of expression evaluation:

- **Parenthesis**: Expressions that are enclosed in parenthesis are always evaluated first.

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

- **Precedence groups**: Operators (listed in Table 4-1) are divided into four precedence groups. When the order of expression evaluation is not determined by parenthesis, the highest-precedence operation is evaluated first.

  Example: 8 + 4/2 = 10 (4/2 is evaluated first)

- **Left-to-right evaluation**: When parenthesis and precedence groups do not determine the order of of expression evaluation, the expressions are evaluated from left to right. (Note that the highest precedence group is evaluated from right to left.)

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

4.6.1 Parentheses in Expressions

Parentheses alter the order of expression evaluation. Parentheses can be nested; the portion of an expression within the innermost parentheses is evaluated first, then the next most innermost pair is evaluated, etc. When all parenthetical expressions have been evaluated, the expression is evaluated from left to right. Parenthetical expressions at the same nesting level are evaluated simultaneously.

The expression **LAB1+((4+3)*7)** is evaluated as follows:

1) Add 4 to 3
2) Multiply 7 by 7
3) Add the value of LAB1 to 49

4.6.2 Operators

Table 4-1 lists the operators than can be used in expressions. They are listed according to precedence group.

Table 4-1. Operators

Group 1 (Highest Precedence) Right-to-Left Evaluation		Group 3 Left-to-Right Evaluation	
+	Unary plus (positive expression)	+	Addition
-	Unary minus (negative expression)	-	Subtraction
~	(COM) 1s complement	|	(OR) Bitwise OR
!	(NOT) Logical NOT (if expr. = 0, 1 is returned, else 0 is returned)	^	(XOR) Bitwise exclusive OR
HI	Right-shift MSbyte into LSbyte, zero fill MSbyte		
LO	AND with 0FFh		
Group 2 Left-to-Right Evaluation		**Group 4 (Relational Operators) Left-to-Right Evaluation**	
*	Multiplication	<	Less than
/	Division	>	Greater than
%	(MOD) Modulo	<=	Less than or equal to
<<	(SHL) Shift left	>=	Greater than or equal to
>>	(SHR) Shift right	=	Equal to
&	(AND) Bitwise AND	<>	Not equal to

Note: Operators in parenthesis indicate an alternate form.

4.6.3 Expression Overflow or Underflow

The assembler checks for overflow and underflow conditions when arithmetic operations are performed at assembly time. The assembler will issue a `Value Truncated` warning whenever an overflow or underflow occurs. The assembler **does not** check for overflow or underflow conditions when multiplication is used within an expression. Examples where a warning message is issued include:

80000000H - 12

4294967290 + 8

4.6.4 Relocatable Symbols and Legal Expressions

Table 4-2 summarizes valid operations on absolute and relocatable 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 or registers that have been defined as global with the .global or .globreg directives can also be used in expressions; In Table 4-2, these symbols and registers are referred to as *external*. Relocatable registers can be used in expressions; the addresses of these registers are relocatable with respect to the .reg section unless they have been declared as external.

Table 4-2. Expressions with Absolute and Relocatable Symbols

A is...	B is...	Results of A+B are...	Results of A-B are...
absolute	absolute	absolute	absolute
absolute	external	external	**illegal**
absolute	relocatable	relocatable	**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.

Examples of legal expressions that use relocatable symbols include:

`blue+1` The value of this expression is the sum of the value of symbol `blue` plus 1. This value is legal and the same type as `blue` (`blue` can be either an absolute or a relocatable symbol).

`GREEN-4` The value of this expression is the result of subtracting 4 from the value of symbol `GREEN`. This value is legal and of the same type as `GREEN` (`GREEN` can be an absolute or a relocatable symbol).

`2*16+red` The value of this expression is the sum of the value of symbol `red` plus the product of 2 times 16. This value is legal and of the same type as `red` (`red` can be an absolute or a relocatable symbol).

`440/2-RED` The value of this expression is the result of dividing 440 by 2 and subtracting the value of symbol `RED` from the quotient (`RED` must be absolute for this to be a legal expression).

`label1-label2` The value of this expression is the difference between the addresses of the two labels. This expressions is only legal if `label1` and `label2` are defined in the same section; in this case, the expression will be absolute.

4.6.5 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.

An example of a well-defined expression is:

`1000h+X` Where X has been previously defined as an absolute symbol.

4.6.6 Conditional Expressions

The assembler supports three directives (.if, .else, and .endif) that provide conditional assembly. The operand of the .if directive is an expression. Operators can be used in an .if expression include:

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

These unsigned operations have the lowest precedence of any operations; however, each has the same precedence within the group, so they are evaluated left to right.

The .if expression evaluates to 1 if true and 0 if false. If the expression evaluates to *true*, then one block of code is assembled; if the expression evaluates to *false*, then another block of code may be assembled.

4.6.7 Examples of Expressions

These examples use five symbols that are defined as follows:

```
            .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
intern_3:                      ; Relocatable, defined in current module
```

- **Example 1:**

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

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

- **Example 2:**

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

  <relocatable symbol> <additive operator> <absolute symbol>

  or

  <absolute value>

  An additive operator is + or -, but not / or *. Unary operators can only be applied 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 statements that follow it are invalid.

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

- **Example 3:**

  In the following examples, the first statement is legal because 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, A ; Legal
MOV   intern_1 + intern_2 + extern_1, A ; 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_2 - intern_2, A     ; Illegal
```

4.7 Addressing Modes

The TMS370 assembler supports seven addressing modes, which are grouped into two classes:

- Direct addressing modes support 8-bit operands.
- Extended addressing modes support 16-bit operands.

Table 4-3 summarizes these addressing modes.

Table 4-3. Addressing Modes

Category	Addressing Mode	Example

Direct	Single register	`label: DEC B`
		`INC R45`
		`CLR R23`

	Register/peripheral file	`label: MOV B,A`
		`ADD A,r17`
		`XOR A,P17`
		`CMP r23,r73`

	Immediate	`label: AND #0C5h,R55`
		`AND #VALUE,R32`
		`BTJO #0D6h,r80,label`

	Program counter relative	`label: JMP label`
		`DJNZ A,label`
		`BTJO B,P7,label`

Extended	Direct memory	`label: MOV 0F3D4h,A` `CMP label,A`
	Register file indirect	`label: MOV @R255,A`
	Indexed	`label: BR label(B)`

4.8 Source Listings

The source listing shows the source statements and the object code they produce.

Each page of the source listing has a header line and a title line at the top. Any title supplied by a .title directive is printed on the title line. If the .title directive is not used, the title line is left blank. A page number is printed to the right of the title. The printer inserts a blank line below the title line and prints a line for each source statement listed.

Each line in the source file produces a line in the listing file that contains an SPC value, the object code assembled, a source statement number, and the source statement. A source statement may produce more than one byte of object code.

```
1    2      3       4
0072 5201   0155    PWRITE0   MOV   #1,B   ;UCR value =  1 (program 0s)
```

Field 1 *Section Program Counter.* This field contains the section program counter value (hexadecimal). Each section (.text, .data, .bss, .reg, and named sections) maintains a separate section program counter. Not all directives affect the section program counter; those directives that do not affect it leave the SPC field unchanged.

On source lines that contain a .equ directive, this field contains the equated value instead of the SPC value.

Field 2 *Object Code.* This field contains the hexadecimal representation of the object code (5201_{16} in this example). All machine instructions use this field to list object code.

Field 3 *Source Statement Number.* The source statement number is a four-digit decimal number. Source lines are numbered in the order in which they appear in the source file, including those source lines that are not listed (.title, .list, .nolist, and .page directives are not listed; source lines between a .nolist directive and a .list directive are not listed). The difference between two consecutive source line numbers indicates the number of source lines entered but not listed. Source lines generated by a macro call or a .copy directive, however, are renumbered starting at 0001. The original sequence continues after the copying or macro expansion is complete. The assembler precedes the line number of included files with a letter code to identify the level of nesting. An **A** indicates the first level, **B** indicates a second level, etc. Macro expansion lines are preceded by a # symbol.

Field 4 *Source Statement Field.* This field contains the characters of the source statement as they were scanned by the assembler. The maximum line length accepted by the assembler is 200 characters. Spacing in this field is determined by the spacing in the source statement.

```
common.asm TMS370 ASSEMBLER Version 3.00 Mon May  4 17:42:06 1987        Pg 1


 0000              0001 ;================ Common Routines ================
 0000              0002
 0000              0003 ;*************************************************
 0000              0004 ; Get address from keyboard input.
 0000              0005 ; Return in INPUT+2,3
 0000              0006 ;*************************************************
 0000              0007
 0000              0008 getaddr:                        ;returns a 16 bit value
 0000              0009          CLR    TEMP1           ;temp 1 = number of chars
 0000              0010          CLR    INPUTBUF        ;clear the 4 address
 0000              0011          CLR    INPUTBUF+1      ;buffer bytes
 0000              0012          CLR    INPUTBUF+2
 0000              0013          CLR    INPUTBUF+3
 0000              0014 GETCH:   TRAP   RXCHAR          ;go fetch a character
 0000 2D**         0015          CMP    #SPACE,A        ;is it a space?
 0002 02**         0016          JEQ    GAEXIT          ;if so exit the routine
 0004 2D**         0017          CMP    #CR,A           ;also exit on a CR
 0006 02**         0018          JEQ    GAEXIT
 0008 2D2D         0019          CMP    #'-',A
 000A 02**         0020          JEQ    GAEXIT
 000C 2D51         0021          CMP    #'Q',A
 000E 02**         0022          JEQ    GAEXIT
 0010              0023          TRAP   HEX             ;is it a valid hex char
 0010 01EE         0024          JN     GETCH
 0012              0025          INC    TEMP1
 0012              0026          MOV    INPUTBUF+1,INPUTBUF
 0012              0027          MOV    INPUTBUF+2,INPUTBUF+1
 0012              0028          MOV    INPUTBUF+3,INPUTBUF+2
 0012 8B****       0029          MOV    A,INPUTBUF+3
 0015 8A****       0030          MOV    LastRX,A
 0018              0031          TRAP   TXCHAR
 0018 00E6         0032          JMP    GETCH
 001A              0033 GAEXIT:
```

Figure 4-2. Sample Assembler Listing

4.9 Cross-Reference Listings

If you use the -x option when you invoke the assembler, it will print a cross-reference listing following the source listing.

```
LABEL                DEFS     REFS                    PAGE 0003

MAXADDR              0009     0039  0043
MAXINT               0010     0008  0011
MININT               0011     0008  0043
```

Figure 4-3. Cross-Reference Listing Format

- The **label** column contains each symbol that was defined or referenced during the assembly.
- The **definition** column contains the statement number in which the symbol is defined. This column is blank for undefined symbols.
- The **reference** column lists the line numbers of statements that reference the symbol. A blank in this column indicates that the symbol was never used.

5. Assembler Directives

Assembler directives supply program data and control the assembly process. Assembler directives allow you to:

- Assemble code and data into specified sections
- Control the appearance of listings
- Initialize constants
- Use conditional assembly
- Define global variables
- Specify libraries that the assembler can obtain macros from

This section is divided into two parts. The directives can be easily categorized by function, and the first part of this section describes the directives according to function. The second part of this section is a reference; the directives are presented in alphabetical order. You will find the following topics in this section:

5.1 Directives Summary

Table 5-1 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 syntax of the directives.

Table 5-1. Directives Summary

Sections Directives	
Mnemonic and Syntax	**Description**
.bss *name[,size]*	Assemble into the .bss (uninitialized data) section
.reg *name[,size]*	Assemble into the .reg section
.regpair *name[,size]*	Assemble into the .reg section
.data *[address]*	Assemble into the .data (initialized data) section
.sect *"name"[,address]*	Assemble into a named (initialized) section
.text *[address]*	Assemble into the .text (executable code) section
Directives that Initialize Constants	
Mnemonic and Syntax	**Description**
.block *size*	Reserve *size* amount of space in the current section
.byte *value 1[,...,value n]*	Initialize 1 or more successive bytes in the current section
.string *"string"*	Initialize a text string
.word *value 1[,...,value n]*	Initialize 1 or more successive words in the current section
Directives that Define Symbols	
Mnemonic and Syntax	**Description**
symbol **.dbit** *bit number,register*	Associate a symbol with a specific bit in a register
symbol **.equ** *value*	Initialize an assembly-time constant
.newblock	"Undefine" local labels
Directives that Format the Output Listing	
Mnemonic and Syntax	**Description**
.length *page length*	Set the page length of the source listing
.list	Restart the source listing
.mlist	Allow macro listings (default)
.mnolist	Inhibit macro listings
.nolist	Stop the source listing
.page	Eject a page in the source listing
.title *"string"*	Print a title in the listing page heading
.width *width*	Set the page width of the source listing

Table 5-1. Directives Summary (Concluded)

Conditional Assembly Directives	
Mnemonic and Syntax	**Description**
.if *expression*	Begin conditional assembly
.else	Optional conditional assembly
.endif	End conditional assembly
Directives that Reference Other Files	
Mnemonic and Syntax	**Description**
.include *"filename"*	Include source statements from another file
.global *symbol 1[,...,symbol n]*	Declare one or more external symbols
.globreg *relocatable register 1 [,...,relocatable register n]*	Declare external relocatable register symbols
.mlib *"filename"*	Supply a macro library name
Miscellaneous Directives	
Mnemonic and Syntax	**Description**
.end	Program end
.setsect	Produced by absolute lister – see Section 10
.setsym	Produced by absolute lister – see Section 10

5.2 Sections Directives

Section 3 discusses COFF sections in detail. The assembler has six directives that associate the various portions of an assembly language program with the appropriate sections:

- The **.bss** directive reserves space in the .bss section for uninitialized data (variables).
- The **.reg** and **.regpair** directives reserve space in the .reg section for relocatable registers. This is an uninitialized section; it must be allocated into the first 256 locations of RAM (the register file).
- The **.text** directive identifies portions of code in the .text section. The .text section usually contains executable code.
- The **.data** directive identifies portions of code in the .data section. The .data section usually contains initialized data.
- The **.sect** directive names a special-purpose section, and associates subsequent code or data with that section.

Figure 5-1 shows how you can use sections directives to associate code and data with the proper sections. This is an output listing; column 1 shows the section program counter (SPC) and column 3 shows the line numbers. The SPC indicates the addresses of code and data in the current section. When code is first assembled into a section, the address in the SPC is 0. The .text, .data, and .sect directives each have an optional address parameter that allows you to specify a different starting address for a section. This parameter is only useful for improving listing readability; it does not affect the final allocation of a section. When you resume assembling into a section, its SPC will resume counting as if there had been no intervening code.

Note that the .bss, .reg, and .regpair directives do not end the current section and begin a new section; they reserve the specified amount of space, and then the assembler returns control to the current section.

After the code in Figure 5-1 is assembled, the sections will contain the following:

.text	Contains bytes 1, 2, 3, 4, 5, and 6
.data	Contains bytes 7, 8, 9, 10, and 11
var_defs	Contains bytes 11, 12, 13 and 14
.bss	Reserves 12 bytes for uninitialized data
.reg	Reserves 8 bytes for relocatable register data

```
0000                0001 ;***********************************************
0000                0002 ;*   Start assembling into the .text section   *
0000                0003 ;***********************************************
0000                0004         .text
0000 0102           0005         .byte  1,2
0002 0304           0006         .byte  3,4
0004                0007
0004                0008 ;***********************************************
0004                0009 ;*   Start assembling into the .data section   *
0004                0010 ;***********************************************
0004                0011         .data
0000 070809         0012         .byte  7,8,9
0003                0013
0003                0014 ;***********************************************
0003                0015 ;*  Define a section named var_defs and begin  *
0003                0016 ;*  assembling code into it                    *
0003                0017 ;***********************************************
0003                0018         .sect  "var_defs"
0000 0B0C           0019         .byte  11,12
0002                0020
0002                0021 ;***********************************************
0002                0022 ;*  Continue assembling into .text             *
0002                0023 ;***********************************************
0002                0024         .text
0004 05             0025         .byte  5
0000                0026         .bss   sym,12      ; Reserve 12 bytes in .bss
0005 06             0027         .byte  6           ; Still in .text
0006                0028
0006                0029 ;***********************************************
0006                0030 ;*  Continue assembling into .data             *
0006                0031 ;***********************************************
0006                0032         .data
0003 0A             0033         .byte  10
0000                0034         .reg   RR1,8       ; Reserve 8 bytes in .reg
0004 0B             0035         .byte  11          ; Still in .data
0005                0036
0005                0037 ;***********************************************
0005                0038 ;*  Continue assembling into var_defs          *
0005                0039 ;***********************************************
0005                0040         .sect  "var_defs"
0002 0D0E           0041         .byte  13,14
```

Figure 5-1. Sections Directives

5.3 Directives that Initialize Constants

There are several directives that you can use to initialize constants:

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

  Figure 5-2 shows an example of the .block directive; assume the following code has been assembled:

```
                        .
                        .
                        .
0154                0003 ;*   Assembling code into .data
0154 000A000B       0004            .word    0Ah,0Bh
0158 0000000000     0005            .block   8
0160 5374696C6C     0006            .string  "Still in .data"
```

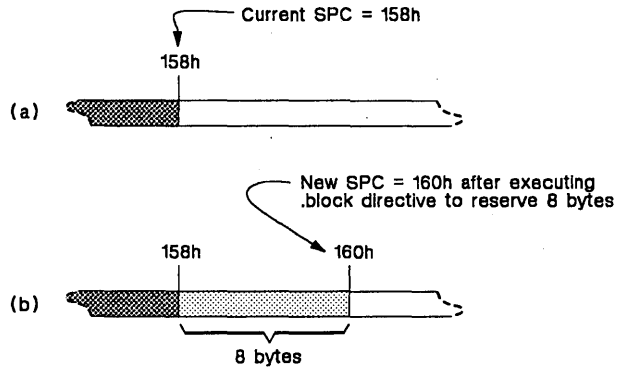
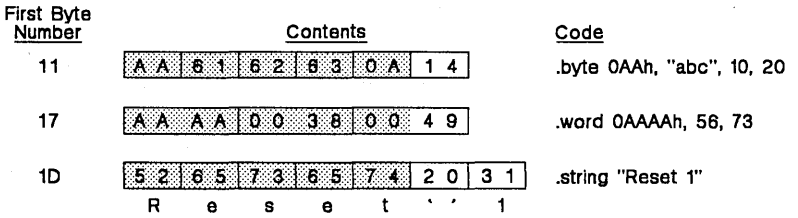


Figure 5-2. An Example of the .block Directive

- The **.byte** directive places 8-bit values into consecutive words of the current section. This directive is similar to .word, except that the width of each value is restricted to 8 bits.
- The **.word** directive places 16-bit values into consecutive locations in the current section.
- The **.string** directive places 8-bit characters from a character string into the current section.

Figure 5-3 compares the .byte, .word, and .string directives. Note that the listing file shows only the first five bytes that are initialized by any of these directives. This example uses .word to reserve three words, but only the most significant byte of the third word is displayed in the listing file. For this example, assume the following code has been assembled:

```
                        .
                        .
                        .
0011 AA6162630A   0003          .byte     0AAh,"abc",10,20
0017 AAAA003800   0004          .word     0AAAAh,56,73
001D 5265736574   0005          .string   "Reset 1"
0024 0F           0006          .byte     15
```



Note: The shaded portion indicates the bytes that are shown in the listing file.

Figure 5-3. Examples of Initialization Directives

5.4 Directives that Define Symbols

The TMS370 family assembler supports several directives that define symbols.

- The **.equ** directive equates a value with a symbol. This type of symbol is known as an *assembly-time constant*; it can be used in the same manner as a constant (for example, in expressions).

  The following example defines a symbol named `bval` and assigns it the value 4. The symbol bval can then be used as a constant.

```
0001           0001   bval     .equ   4
0002 040810    0002            .byte  bval, bval*2, bval+12
```

  Note that the .bss, .reg, and .regpair directives also define symbols, and that labels are symbols; like symbols defined by the .equ directive, these symbols can be used as constants.

- The **.dbit** directive defines a symbol that references a specific bit in a peripheral register. This symbol can then be used as an operand for the the following instructions:

  - CMPBIT
  - SBIT0
  - SBIT1
  - JBIT0
  - JBIT1

- The **.newblock** directive *resets* local labels. Local labels are symbols of the form `$n` (n is a decimal digit); they are defined when they appear in the label field. Local labels are temporary labels that can be used as operands for jump instructions. The .newblock directive limits the scope of local labels by "undefining" them after they're used.

5.5 Directives that Format the Output Listing

There are eight directives that you can use to format the listing file:

- 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 stop 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, respectively, allow and inhibit macro expansion listings.
- The **.page** directive causes a page eject in the output listing.
- The **.title** directive supplies a title that the assembler will print on the first line of each page.

5.6 Conditional Assembly Directives

Three directives allow you to assemble conditional blocks of code:

- The **.if** directive marks the beginning of a conditional block. The .if directive has one parameter, which is an expression. If this expression evaluates to **true** (a nonzero value), then the assembler will assemble the code that follows it (up to an .else or .endif). If this expression evaluates to **false** (0), then the assembler will assemble code that follows an .else (if present) or an .endif (if no .else is present).

- The **.else** directive indicates a block of code that the assembler will assemble if 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, then the assembler will continue with the code that follows the .endif.

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

Four operators can be used in an .if expression:

= Equal
<> Not equal
< Less than
> Greater than

These operations are all unsigned. They have the lowest precedence of any operations. However, they have the same precedence, and are evaluated left to right. They evaluate to 1 if true and 0 if false.

Here is an example of conditional assembly:

```
0001                0001 sym1     .equ    1
0002                0002 sym2     .equ    2
0003                0003 sym3     .equ    3
0004                0004 sym4     .equ    4
0000                0005 If_1:    .if     sym1 < sym2
0000 01             0006          .byte   sym1
0001                0007          .else
0001                0008          .byte   sym2
0001                0009          .endif
0001                0010 If_2:    .if     sym1 + sym2 = sym3
0001 0102           0011          .byte   sym1,sym2
0003                0012          .else
0003                0013          .byte   sym3
0003                0014          .endif
0003                0015 If_3:    .if     sym3 <> sym4 - sym2
0003 03             0016          .byte   sym3
0004                0017          .else
0004                0018          .byte   sym4
0004                0019          .endif
```

5.7 Directives that Reference Other Files

These directives supply information for or about other files.

- The **.include** directive tells the assembler to begin reading source statements from another file. When the assembler is done reading the source statements in the .include file, it will resume reading source statements from the current file.

- The **.global** and **.globreg** directives declare a symbol or a relocatable register symbol, respectively, to be external. This makes the symbol (or relocatable register symbol) available to other modules at link time. If the symbol is *defined in the current module,* it can be used in other modules. If the symbol is *used but not defined in the current module,* the linker can look for its definition in another module.

  This allows you to assemble program modules separately and link them together to form a single executable program.

- 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 will then be able to search for it in the specified macro library.

5.8 Directives Reference

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

Syntax **.block** <size>

Description The .block directive reserves *size* number of bytes in the current section and fills them with 0s. The section program counter (SPC) is incremented to point to the byte that follows the reserved block.

The .block directive is equivalent to *size* number of `.byte 0` directives.

Example Reserve 100 0-filled bytes in the .text section. Note that the SPC equals 0Eh before the .block directive is assembled; after the .block directive is assembled, the SPC is incremented to equal 072h.

```
0000                    0001 ;*****************************************
0000                    0002 ;*       Begin assembling into .text     *
0000                    0003 ;*****************************************
0000                    0004           .text
0000 000A000B           0005           .word    0Ah,0Bh
0004 5265676973         0006           .string  "Register A"
000E                    0007
000E                    0008 ;*****************************************
000E                    0009 ;*  Reserve a block of 100 bytes in .text  *
000E                    0010 ;*****************************************
000E 0000000000         0011           .block   100
0072                    0012
0072 000C               0013           .word    0Ch          ; Still in .text
```

Syntax **.bss** <name>,<size>

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 a required parameter. It is an absolute expression that specifies the number of bytes that will be allocated. There is no default size for this directive.

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, and .regpair), however, *do not affect the current section*. The assembler will assemble the .bss, .reg, or .regpair directive and then resume assembling code into the same section.

For more information about COFF sections, see Section 3.

Example This example uses the .bss directive 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.

```
0000                    0001 ;****************************************
0000                    0002 ;*   Begin assembling into .text section   *
0000                    0003 ;****************************************
0000                    0004       .text
0000 420001             0005       MOV     R0,R1
0003                    0006
0003                    0007 ;****************************************
0003                    0008 ;*   Allocate 100 bytes in .bss section   *
0003                    0009 ;****************************************
0000                    0010       .bss    array,100
0003                    0011
0003 420102             0012       MOV     R1,R2     ; Assembled into .text
0006                    0013
0006                    0014 ;****************************************
0006                    0015 ;*     Allocate 1 byte in .bss section    *
0006                    0016 ;****************************************
0064                    0017       .bss    dflag,1
0006                    0018
0006 8A0064             0019       MOV     dflag,R0 ; Assembled into .text
0009                    0020
0009                    0021 ;****************************************
0009                    0022 ;*      Declare external .bss symbol      *
0009                    0023 ;****************************************
0009                    0024       .global array    ; Still in .text
```

Syntax **.byte** <value 1>[,...,<value *n*>]

Description The .byte directive places one or more 8-bit values into consecutive bytes in the current section. Each value can be either:

- An expression which the assembler will evaluate and treat as an 8-bit signed number.
- A character string enclosed in double quotes. Each character represents a separate value.

The assembler will truncate values that are greater than 8 bits. Each character in a string is counted as a separate operand. You can use as many values as will fit on a line, but the assembler will only show a maximum of five bytes per line in the listing file.

If you use a label, it will point to the location at which the assembler places the first byte.

Example This example places the 8-bit values 10, -1, 97, 98, 99, and 97 into six consecutive bytes in memory. The label `strx` has the value 0h, which is the location of the first initialized byte.

```
0000 0AFF616263   0001 strx:    .byte     10,-1,"abc",'a'
```

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 only be used 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.

Section 3 provides a detailed explanation about COFF sections.

> **Note:**
>
> The assembler assumes that .text is the default section. Therefore, at the beginning of an assembly, the assembler will assemble code into the .text section unless you specify an explicit section control directive.

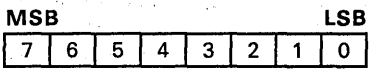
Example This example shows how you can use the .text and .data directives to swap between sections.

```
0000                0001 ;*****************************************
0000                0002 ;*     Begin assembling into .text       *
0000                0003 ;*****************************************
0000                0004        .text
0000 420001         0005        MOV    R0,R1    ; Assembled into .text
0003                0006
0003                0007 ;*****************************************
0003                0008 ;*     Begin assembling into .data       *
0003                0009 ;*****************************************
0003                0010 table: .data
0000 FFFF           0011        .word  -1       ; Assembled into .data
0002 FF             0012        .byte  0ffh     ; Assembled into .data
0003 420001         0013        MOV    R0,R1    ; Assembled into .data
0006                0014
0006                0015 ;*****************************************
0006                0016 ;*   Resume assembling into .text at 03h   *
0006                0017 ;*****************************************
0006                0018        .text
0003 0038004E       0019        .word  56,78
0007                0020
0007                0021 ;*****************************************
0007                0022 ;*   Resume assembling into .data at 06h   *
0007                0023 ;*****************************************
0007                0024        .data
0006 FD             0025        LDSP
```

Syntax <symbol> **.dbit** <bit number>,<register>

Description The .dbit directive assigns a name to a specific bit in a register. The *symbol* is the name that the assembler assigns to the bit; it must appear in the label field. The *register* must be a R0–R255, P0–P255, or a symbol that has been equated to one of these registers. The *bit number* is a number in the range 0–7, and indicates a particular bit in the specified register.

Note that register bits are numbered like this:



The TMS370 assembler supports several instructions that operate on single bits. Before you can execute these instructions, you must use the .dbit directive to name the bit that will be operated on. These instructions include:

CMPBIT Complement a specified bit.
JBIT0 Jump if a specified bit equals 0.
JBIT1 Jump if a specified bit equals 1.
SBIT0 Set a specified bit to 0.
SBIT1 Set a specified bit to 1.

Example This example sets up bits in the Serial Port Interface Configuration register, which is denoted in this example by the symbol SPCF.

```
0000                    0001 ;***********************************************
0000                    0002 ;* Setup the bits in the Serial Port Interface *
0000                    0003 ;* Configuration register                      *
0000                    0004 ;***********************************************
0000                    0005
0000                    0006 SPCF_S_u:
0000                    0007
0000                    0008 Char0          .dbit   0,SPCF
0000                    0009 Char1          .dbit   1,SPCF
0000                    0010 Char2          .dbit   2,SPCF
0000                    0011 SPBR0          .dbit   3,SPCF
0000                    0012 SPBR1          .dbit   4,SPCF
0000                    0013 SPBR2          .dbit   5,SPCF
0000                    0014 CPOL           .dbit   6,SPCF
0000                    0015 SWRST          .dbit   7,SPCF
0000                    0016
0000                    0017 ;***********************************************
0000                    0018 ;* Setup char bits so that 8 bits are shifted  *
0000                    0019 ;* out per character                            *
0000                    0020 ;***********************************************
0000                    0021            SBIT1   Char0
0000                    0022            SBIT1   Char1
0000                    0023            SBIT1   Char2
0000                    0024
0000                    0025 ;***********************************************
0000                    0026 ;* Setup the serial peripheral bit rate so that*
0000                    0027 ;* the shift clock frequency = system clock/64 *
0000                    0028 ;***********************************************
0000                    0029            SBIT1   SPBR0
0000                    0030            SBIT0   SPBR1
0000                    0031            SBIT1   SPBR2
```

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 will ignore any source statements that follow the .end directive.

> **Note:**
>
> This directive has the same effect as an end-of-file.

Example This example shows how the .end directive terminates assembly. If any source statements followed the .end directive, they would not be assembled.

```
0000                    0001 Text_Start:    .text
0000 0A                 0002                .byte    0Ah
0001 AAAA               0003                .word    0AAAAh
0003 4141414141         0004                .string  "AAAAAAAA"
000B                    0005                .end
```

Syntax <symbol> **.equ** <value>

Description The .equ directive equates a 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 *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 or global and defined in another 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 SPC 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 .equ.

```
0000                0001 ;**************************************
0000                0002 ;** Equate symbol FP to register R3   **
0000                0003 ;** and use it instead of R3         **
0000                0004 ;**************************************
0003                0005 FP      .equ   R3
0000 9A03           0006         MOV    @FP,R0
0002                0007
0002                0008 ;**************************************
0002                0009 ;** Set symbol count to an integer    **
0002                0010 ;** expression and use it as an      **
0002                0011 ;** immediate operand                **
0002                0012 ;**************************************
0035                0013 count   .equ   100/2 + 3
0002 723500         0014         MOV    #count,R0
0005                0015
0005                0016 ;**************************************
0005                0017 ;** Set symbol symtab to relocatable  **
0005                0018 ;** expression                       **
0005                0019 ;**************************************
0005 000A           0020 label   .word  10
0006                0021 symtab  .equ   label+1
0007 0006           0022         .word  symtab
0009                0023
0009                0024 ;**************************************
0009                0025 ;** Set symbol nsyms equal to another **
0009                0026 ;** symbol (count) and use it instead **
0009                0027 ;** of count                          **
0009                0028 ;**************************************
0035                0029 nsyms   .equ   count
0009 723500         0030         MOV    #nsyms,R0
```

Syntax **.global** <symbol 1>[,...,<symbol *n*>]

Description The .global directive defines a symbol that can be referenced externally. 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 .equ or .bss directive. When a symbol is declared as global, it becomes a *16-bit* value. As with all symbols, if a global symbol is defined more than once, the linker will issue a multiple-definition error. A symbol may be declared global for two reasons:

1) If the symbol is *not defined in the current source module* (this includes macro and include files), then the .global 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 will look for the symbol's definition in other modules.

2) If the symbol *is defined in the current module*, then the .global directive declares that the symbol and its definition can be used externally in other modules. These types of references will be resolved at link time.

A symbol can be equated to a global symbol only if the global symbol is defined before the .equ statement is assembled:

```
        .global x
x:
y:      .equ    x       ; Legal
```

A global symbol cannot be used as an operand in an .equ statement if the global symbol has not been declared:

```
        .global x
y:      .equ    x       ; Illegal
x:
```

Example `file1.asm` declares `Init`, `X`, `Y`, and `Z` as global symbols. `file1.asm` defines `Init`, and `file2.asm` uses it; `file2.asm` defines `X`, `Y`, and `Z`, and `file1.asm` uses them.

file1.asm:

```
0000                    0001 ; File 1
0000                    0002 ; Global symbol defined in this file
0000                    0003         .global  Init
0000                    0004 ; Global symbols defined in other files
0000                    0005         .global  X,Y,Z
0000                    0006
0000                    0007 Init:   .equ     0       ; Symbol definition
0000 C5                 0008         CLR      B
0001 8A0000             0009         MOV      X,A
0004 3800               0010         ADD      A,B
0006 8A0000             0011         MOV      Y,A
0009 3800               0012         ADD      A,B
000B 8A0000             0013         MOV      Z,A
000E 3800               0014         ADD      A,B
```

file2.asm:

```
0000                    0001 ; File 2
0000                    0002 ; Global symbol defined in another file
0000                    0003         .global Init
0000 8A0000             0004         MOV     Init,A
0005                    0005 X       .equ    5
000A                    0006 Y       .equ    10
000F                    0007 Z       .equ    15
```

Syntax **.globreg** <symbol 1>[,...,<symbol *n*>]

Description The .globreg directive defines a register symbol that can be referenced externally. A register symbol is defined by the .reg directive. As with all symbols, if a global register symbol is defined more than once, the linker will issue a multiple-definition error.

A register symbol may be declared global for two reasons:

1) If the register symbol is *not defined in the current source module* (this includes macro and include files), then the .globreg directive tells the assembler that the register symbol is defined in an external module. This prevents the assembler from issuing an unresolved reference error. At link time, the linker will look for the register symbol's definition in other modules.

2) If the register symbol *is defined in the current module*, then the .globreg directive declares that the register symbol and its definition can be used externally in other modules. These types of references will be resolved at link time.

Example The following example defines two register variables, RR0 and RR1, and then declares them global with the .globreg directive.

```
0000                    0001 ;*  Begin assembling into .text
0000                    0002           .text
0000 223F               0003           MOV       #63,A
0002 521C               0004           MOV       #28,B
0004 3800               0005           ADD       A,B
0006                    0006
0006                    0007 ;*  Declare two relocatable registers,
0006                    0008 ;*  RR0 and RR1
0000                    0009           .reg      RR0,3
0004                    0010           .regpair  RR1
0006                    0011
0006                    0012 ;*  Make RR0 and RR1 global
0006                    0013           .globreg  RR0,RR1
```

Syntax

.if <expression>

code to assemble if expression is true (≠ 0)

.else

code to assemble if expression is false (= 0)

.endif

Description

Three directives provide conditional assembly:

- The **.if** directive identifies the beginning of a conditional block. *Expression* is a required parameter. If this expression evaluates to **true** (a nonzero value), then the assembler will assemble the code that follows it (up to an .else or .endif). If this expression evaluates to **false** (0), then the assembler will assemble code that follows an .else (if present) or an .endif (if no .else is present).
- The **.else** directive identifies a block of code that the assembler will assemble if 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, then the assembler will continue with the code that follows the .endif.
- The **.endif** directive terminates a conditional block.

Example

Here are some examples of conditional assembly:

```
0001                0001 sym1    .equ    1
0002                0002 sym2    .equ    2
0003                0003 sym3    .equ    3
0004                0004 sym4    .equ    4
0000                0005 If_1:   .if     sym1 < sym2
0000 01             0006         .byte   sym1
0001                0007         .else
0001                0008         .byte   sym2
0001                0009         .endif
0001                0010 If_2:   .if     sym1 + sym2 = sym3
0001 0102           0011         .byte   sym1,sym2
0003                0012         .else
0003                0013         .byte   sym3
0003                0014         .endif
0003                0015 If_3:   .if     sym3 <> sym4 - sym2
0003 03             0016         .byte   sym3
0004                0017         .else
0004                0018         .byte   sym4
0004                0019         .endif
```

Syntax **.include** "<filename>"

Description The .include directive tells the assembler to read source statements from a different file. *Filename* is the name of a source file, enclosed in double quotes. At the end-of-file for *filename*, the assembler will resume processing source statements from the file or device it was processing before the .include was encountered.

An .include directive may be nested within a file being copied. The assembler limits nesting to eight levels; the host operating system may set additional restrictions. The assembler precedes the line number of copied files with a letter code to identify the source file. An **A** indicates the first file, **B** indicates a second file, etc.

Note that you cannot use the .include directive within a macro definition.

Note that each include file requires additional file space. To do this, place a FILES=nn command in the config.sys file; nn should be the number of assembly language source files plus six.

Example This example reads and assembles source statements from the file byte.asm, and then resumes assembling the current file. Note that the lines that are assembled from the first include file are preceded by the letter **A**, and lines from the second include file are preceded by the letter **B**.

include.asm (source file)

```
    .block   20
    .include "byte.asm"

; Back in original file
    .string  "Done"
```

byte.asm (first include file)

```
; In byte.asm
    .byte    32,1+'A',1+"A"
    .include "word.asm"
; Back in byte.asm
    .byte    67h + 3
```

word.asm (second include file)

```
; In word.asm
    .word    0AABBh,56q
```

Listing file:

```
0000 0000000000    0001          .block     20
0014               0002          .include   "byte.asm"
0014               A0001 ; In byte.asm
0014 204242        A0002          .byte      32,1+'A',1+"A"
0017               A0003          .include   "word.asm"
0017               B0001 ; In word.asm
0017 AABB002E      B0002          .word      0AABBh,56q
001B               A0004 ; Back in byte.asm
001B 6A            A0005          .byte      67h + 3
001C                0003
001C                0004 ; Back in original file
001C 446F6E65       0005          .string    "Done"

*************** SOURCE FILES ***************


     ID   FILENAME

          include.asm
      A   byte.asm
      B   word.asm
```

Syntax **.length** <page length>

.width <page width>

Description The .length directive sets the page length of the output listing file. The default page length is 60 lines. The maximum page length is 32,767 lines.

The .width directive sets the page width of the output listing file. The default page width is 80 characters. The maximum page width is 200 characters. Comments and other portions of a source statement that extend beyond the page width are truncated in the listing.

Example This example sets the page length and the page width to various values.

```
length.asm     TMS370 ASSEMBLER Version 2.96  Wed Apr 29 17:40:49  Page    1
  ***   Length and Width   ***


 0000              0001           .title    "***   Length and Width   ***"
 0000              0002
 0000              0003 ;*****************************************************
 0000              0004 ;*****************************************************
 0000              0005 ;*        The page length is limited to 60           *
 0000              0006 ;*        lines per page. The page width is          *
 0000              0007 ;*        limited to 60 characters per line.         *
 0000              0008 ;*****************************************************
 0000              0009 ;*****************************************************
 0000              0010           .length   60
 0000              0011           .width    60
 0000              0012
 0000              0013 ;*****************************************************
 0000              0014 ;*****************************************************
 0000              0015 ;*        The page length is limited to 50           *
 0000              0016 ;*        lines per page. The page width is          *
 0000              0017 ;*        limited to 150 characters per line.        *
 0000              0018 ;*****************************************************
 0000              0019 ;*****************************************************
 0000              0020           .length   50
 0000              0021           .width    150
```

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. The .nolist directive can be used to reduce assembly time and the size of the source listing.

The assembler prints the .nolist directive, but it does not print the .list directive or the directives that appear after a .nolist directive. The assembler continues to increment the line counter and the SPC for the source statements that are not listed.

By default, the assembler acts as if a .list directive was assembled at the beginning of a program.

> **Note:**
>
> If you don't request a listing file when you invoke the assembler, the assembler will ignore the .list directive.

Example This example uses the .include directive to insert source statements from another file. The first time this directive is invoked, the assembler lists the copied source lines in the listing file. The second time this directive is invoked, the assembler does not list the copied source lines because a .nolist directive was assembled.

Source file:

```
        .include "file1.asm"
        .nolist
        .include "file1.asm"
        .list
; Back in original file
        .string "Done"
```

Listing file:

```
0000                 0001         .include "file1.asm"
0000                A0001 ;********** In file1.asm **********
0000 0AFF616263     A0002         .byte     10,-1,"abc",'a'
0006                 0002         .nolist
000C                 0005 ; Back in original file
000C 446F6E65        0006         .string "Done"
```

Syntax **.mlib** "<filename>"

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 an archive) by the archiver. Each file in a macro library may contain one macro definition that corresponds to the name of the file.

Filename specifies a source file that is named according to host operating system conventions. The name must be enclosed in double quotes.

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 files within the library 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. Only macros that are actually called from the library are extracted.

Macros that are defined in libraries are expanded in the same manner as macros that are defined in a source module; however, such macro definitions do not appear in the listing file because they aren't part of the source file.

Example This example creates a macro library that defines two macros, `incqw` and `decqw`. The file `incqw.asm` contains the definition of `incqw`, and `decqw.asm` contains the definition of `decqw`.

incqw.asm

```
; Macro for incrementing
; 32-bit word
incqw   .MACRO    reg
        .newblock
        incw      #1,:reg:
        jnc       $1
        incw      #1,:reg:-2
$1
        .endm
```

decqw.asm

```
; Macro for decrementing
; 32-bit word
decqw   .MACRO    reg
        .newblock
        incw      #-1,:reg:
        jc        $1
        incw      #-1,:reg:-2
$1
        .endm
```

Use the archiver to create a macro library:

```
ar370 -a mac incqw.asm decqw.asm
```

Now you can use the .mlib directive to reference the macro library and define the `incqw` and `decqw` macros:

```
.mlib   "mac.lib"
incqw   r8          ; Macro call
decqw   r8          ; Macro call
```

Syntax **.mlist**

.mnolist

There are two directives that provide you with the ability to control the listing of macro expansions in the listing file:

- The **.mlist** directive allows macro expansions in the listing file.
- The **.mnolist** directive inhibits macro expansions in the listing file.

By default, macro expansions are printed in the listing.

Example This example defines a macro named `str_3`. The first time `str_3` 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 expansion is again listed because a .mlist directive was assembled (effectively cancelling the .mnolist).

Notice that the source statements generated by a macro expansion are precede with a # character.

```
0000                0001 str_3      .MACRO      parm1,parm2,parm3
0000                0002            .string     :parm1:
0000                0003            .string     :parm2:
0000                0004            .string     :parm3:
0000                0005            .ENDM
0000                0006
0000                0007            str_3       "red","green","blue"
0000 726564         0001 #             .string     "red"
0003 677265656E     0002 #             .string     "green"
0008 626C7565       0003 #             .string     "blue"
000C                0008            .mnolist
000C                0009            str_3       "Socrates","Plato","Aristotle"
0022                0010            .mlist
0022                0011            str_3       "Huron","Superior","Michigan"
0022 4875726F6E     0001 #             .string     "Huron"
0027 5375706572     0002 #             .string     "Superior"
002F 4D69636869     0003 #             .string     "Michigan"
```

Syntax **.newblock**

Description The .newblock directive "undefines" any local labels that are 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 word. Unlike other labels, local labels cannot be used in expressions; they can only be used as the operand in 8-bit jump instructions.

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 named sections 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 declares the local label `$1`, resets it, and then declares it again.

```
0000 420203       0001 Label1:   mov    r2,r3
0003 06**         0002           jnz    $1
0005 72FF03       0003           mov    #-1,r3
0008 1D03         0004 $1        cmp    r3,A
000A              0005           .newblock      ; Undefine $1
000A 06**         0006           jne    $1
000C D303         0007           inc    r3
000E 480304       0008 $1        add    r3,r4
```

Syntax **.page**

Description The .page directive produces a page eject in the listing file. Using the .page directive to divide the source listing into logical divisions makes the listing easier to read.

Example This example causes the assembler to begin a new page of the source listing.

```
page.asm        TMS370 ASSEMBLER Version 2.96  Wed Apr 29 17:48:00  Page   1
    ** The .title directive works **


 0000               0001             .title   "** The .title directive works **"
                                        .
                                        .
                                        .
                                        .
                                        .
 0000               0060             .page

page.asm        TMS370 ASSEMBLER Version 2.96  Wed Apr 29 17:48:00  Page   2
    ** The .title directive works **
                                        .
                                        .
                                        .
                                        .
                                        .
```

Syntax

.reg <name>[,<size>]

.regpair <name>[,<size>]

Description

The .reg and .regpair directives allocate a block of memory in the register file that can be used for relocatable registers. There are two differences between these directives:

- The register symbol defined by the .reg directive points to the *first* byte reserved; the register symbol defined by the .regpair directive points to the *last* byte reserved.
- The default size for .reg is *one* byte; the default size for .regpair is *two* bytes.

Relocatable registers allow you to use registers whose addresses will be determined at link time. The linker maps a register to the first available register location, so you do not have to explicitly specify which register location you want to use. The linker begins allocating registers starting with R2; registers A and B (R0 and R1) cannot be used as relocatble registers. Note that you can use the .globreg directive to make a relocatable register symbol global; this allows you to refer to the same register in different source modules.

- The *name* is a required parameter. It defines a relocatable register symbol, which can be used like any other register symbol.
  - When you use the .reg directive, the symbol points to the **first** byte that is reserved.
  - When you use the .regpair directive, the symbol points to the **last** byte that is reserved.
- The *size* is an optional parameter. It is an absolute expression that specifies the number of bytes that will be allocated.
  - If you do not specify a size for the .reg directive, it will reserve **one byte** in the .reg section.
  - If you do not specify a size for the .regpair directive, it will reserve **two bytes** in the .reg 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, and .regpair), however, *do not affect the current section*. The assembler will assemble the .bss, .reg, or .regpair directive and then resume assembling code into the same section.

For more information about COFF sections, see Section 3.

Example This example uses the .reg and .regpair directives to allocate register space for five relocatable registers (RR0, RR1, RR2, RR3, and RR4). Figure 5-4 illustrates the effects of these .reg and .regpair directives on the .reg section. Note that these directives occur within a block of code that is assembled into the .text section; however, they do not affect the .text section.

```
0000                0001 ;***********************************************
0000                0002 ;*       Begin assembling code into .text       *
0000                0003 ;***********************************************
0000                0004
0000                0005          .text
0000 00040005       0006          .word    4,5      ; Assembled into .text
0004 06             0007          .byte    6        ; Assembled into .text
0005 00070008       0008          .word    7,8      ; Assembled into .text
0009                0009
0000                0010          .reg     RR0      ; Assembled into .reg
0002                0011          .regpair RR1      ; Assembled into .reg
0003                0012          .reg     RR2,3    ; Assembled into .reg
0008                0013          .regpair RR3,3    ; Assembled into .reg
0009                0014          .reg     RR4      ; Assembled into .reg
0009                0015
0009 417578         0016          .string  "Aux"    ; Assembled into .text
```

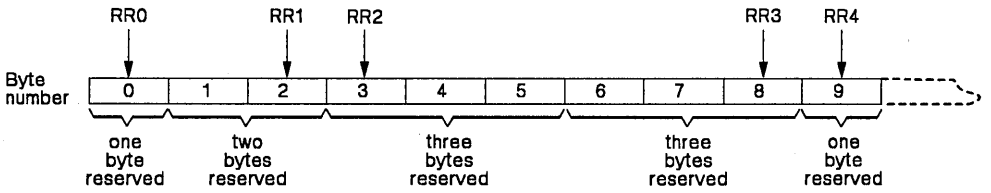


Figure 5-4. Examples of the .reg and .regpair Directives

Syntax .sect "<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 *name* is a required parameter. It is significant to 8 characters and must be enclosed in double quotes. When used, the label points to the location in the section *name*.
- The *address* is an optional parameter that specifies a 16-bit address. It can only be used 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.

Section 3 provides additional information about named sections.

Example This example defines two named sections and assembles code into them.

```
0000                    0001 ;***********************************************
0000                    0002 ;**    Begin assembling into .text section      **
0000                    0003 ;***********************************************
0000                    0004         .text
0000 420102             0005         MOV      R1,R2    ; Assembled into .text
0003 420304             0006         MOV      R3,R4    ; Assembled into .text
0006                    0007
0006                    0008 ;***********************************************
0006                    0009 ;**    Begin assembling into Sym_Defs section   **
0006                    0010 ;***********************************************
0006                    0011         .sect    "Sym_Defs"
0000 013A               0012         .word    314
0002 0F                 0013         .byte    0Fh
0003 420506             0014         MOV      R5,R6
0006                    0015
0006                    0016 ;***********************************************
0006                    0017 ;**    Begin assembling into addi section       **
0006                    0018 ;***********************************************
0006                    0019         .sect    "addi",16
001F                    0020 Jan:    .equ     31
001C                    0021 Feb:    .equ     28
001F                    0022 Mar:    .equ     31
0010 221F               0023         MOV      #Jan,A
0012 521C               0024         MOV      #Feb,B
0014 3800               0025         ADD      A,B
0016                    0026
0016                    0027 ;***********************************************
0016                    0028 ;**    Resume assembling into .text section     **
0016                    0029 ;***********************************************
0016                    0030         .text
0006 421415             0031         MOV      R20,R21
0009 0304               0032         .byte    3,4
000B                    0033
000B                    0034 ;***********************************************
000B                    0035 ;**    Resume assembling into addi section      **
000B                    0036 ;***********************************************
000B                    0037
000B                    0038         .sect    "addi"
0016 521F               0039         MOV      #Mar,B
0018 3800               0040         ADD      A,B
001A                    0041
001A                    0042 ;***********************************************
001A                    0043 ;**    Resume assembling into Sym_Defs section  **
001A                    0044 ;***********************************************
001A                    0045
001A                    0046         .sect    "Sym_Defs"
0006 AABB               0047         .word    0aabbh
0008 CCDD               0048         .word    0ccddh
```

Syntax **.string** "<string>"

Description The .string directive places 8-bit characters from a character string into the current section. An operand must be a character string enclosed in double quotes; each character in a string represents a separate byte. Values are packed into words starting with the most significant byte of the word and moving toward the least significant portion as more bytes are added.

You may use only *one* operand per .string directive. Although the assembler will initialize all the bytes necessary to place the string into memory, it will only show the first five bytes in the listing file. Note that the SPC is incremented by the number of bytes that are initialized.

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

Example This example places several strings into memory.

```
0000 41424344      0001 strptr   .string   "ABCD"
0004 3432          0002          .string   "42"
0006 68656C6C6F    0003          .string   "hello"
000B 416D737465    0004          .string   "Amsterdam"
```

Syntax **.text** [address]

Description The .text directive tells the assembler to begin assembling into the .text section, which 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 only be used 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:**
>
> 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 section directives (.text, .data, or .sect).

For more information about COFF sections, see Section 3.

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

```
0000                    0001 ;****************************************
0000                    0002 ;* Begin assembling into .data section *
0000                    0003 ;****************************************
0000                    0004          .data
0000 0506               0005          .byte   5,6
0002                    0006
0002                    0007 ;****************************************
0002                    0008 ;* Begin assembling into .text section *
0002                    0009 ;****************************************
0002                    0010          .text
0000 01                 0011          .byte   1
0001 0203               0012          .byte   2,3
0003                    0013
0003                    0014 ;****************************************
0003                    0015 ;* Resume assembling into .data section *
0003                    0016 ;****************************************
0003                    0017          .data
0002 0708               0018          .byte   7,8
0004                    0019
0004                    0020 ;****************************************
0004                    0021 ;* Resume assembling into .text section *
0004                    0022 ;****************************************
0004                    0023          .text
0003 04                 0024          .byte   4
```

Syntax **.title** "<string>"

Description The .title directive supplies a title that is printed in the heading on each listing page. *String* is a quote-enclosed title of up to 50 characters. If you supply more than 50 characters, the assembler will truncate the string and issue a warning. The assembler prints the title on the page that follows the directive, and on subsequent pages until another .title directive is processed.

Example This example prints the title ** *The .title directive works* ** in the page headings of the source listing.

Source statement:

```
        .title    "** The .title directive works**"
```

Listing file:

```
page.asm        TMS370 ASSEMBLER Version 2.96  Wed Apr 29 17:48:00  Page   1
  ** The .title directive works **


 0000              0001              .title    "** The .title directive works **"
                                       .
                                       .
                                       .
                                       .
                                       .
 0000              0060              .page

page.asm        TMS370 ASSEMBLER Version 2.96  Wed Apr 29 17:48:00  Page   2
  ** The .title directive works **
                                       .
                                       .
                                       .
                                       .
                                       .
```

Syntax **.word** <value 1>[,...,<value n>]

Description The .word directive place one or more values into consecutive words in the current section. Each operand is either an expression or a character string; each character in a character string represents a separate value.

The operand 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 with labels.

You can use up to 100 operands, but they must fit on a single source statement line. Although the assembler will initialize as many bytes as indicated by the .word directive, it will only show up to five bytes in the listing file. The assembler increments the section program counter by the number of bytes it initializes. If you use a label, it will point to the first word that is initialized.

Example This example initializes five words. The symbol `WordX` points to the first word that is initialized.

```
0000 0C804143BE   0001 WordX: .word 3200,1+'AB',-'AF',0F410h,'A'
```

6. TMS370 Instruction Set Summary

This section summarizes the TMS370 family instruction set. Table 6-1 lists the symbols and abbreviations that are used throughout this section.

Table 6-1. Symbols and Abbreviations Used in the Instruction Set Summary

Symbol	Description	Symbol	Description
n1,n2,n3	8-bit integers in the range 0–255. When used without a prefix, they specify an 8-bit signed address.	n1n2	n1 and n2 are concatenated to form a 16-bit signed address.
#n1	8-bit immediate value	#n1n2	16-bit immediate value
Pn1	Specifies the contents of peripheral file location n1 (a memory location from 1000h to 10FFh); Pn2 and Pn3 may also be used	Rn1	Specifies the contents of register n1 (a memory location from 0000h to 00FFh); Rn1 and Rn2 may also be used
@	A prefix that indicates a 16-bit address	@Rn1	A 16-bit address – register Rn1 contains the LSB, register Rn1-1 contains the MSB
A	Contents of the A register (memory location 0000h)	B	Contents of the B register (memory location 0001h)
PC	Program Counter (contains the address of the current instruction)	UPC	Updated Program Counter (points to the next instruction)
PCh	8 MSBs of the PC	PCl	8 LSBs of the PC
SP	Stack Pointer	ST	Status Register (contains the interrupt enable bits and C, N, Z, and V flags)
C	Carry flag	N	Sign flag
V	Overflow/borrow flag	Z	Zero flag

The instruction set summary tables show six types of information for each instruction.

1) Column 1 shows the instruction syntax and lists the possible combinations of operands. If a note is referenced "for all," then the note applies to all the combinations; otherwise, it applies to one combination only.
2) Column 2 lists opcodes as hexadecimal constants; each opcode represents one byte. If the second byte of the field contains a value, then the opcode is two bytes long; if this field is blank, then the opcode is one byte long.
3) Column 3 lists the operands.
4) Column 4 shows how instruction execution affects the C, N, Z, and V status flags.
5) Column 5 lists the number of cycles consumed by instruction execution.
6) Column 6 illustrates the result of instruction execution on the operands.

Instruction Format		Opcode	Operands			Status Flags†				Cycles	Action Description
		1 2	1	2	3	C	N	Z	V		
ADC	B,A	69				x	x	x	x	8	A := B + A + C
	Rn1,A	19	n1			x	x	x	x	7	A := Rn1 + A + C
	Rn1,B	39	n1			x	x	x	x	7	B := Rn1 + B + C
	Rn1,Rn2	49	n1	n2		x	x	x	x	9	Rn2 := Rn1 + Rn2 + C
	#n1,Rn2	79	n1	n2		x	x	x	x	8	Rn2 := n1 + Rn2 + C
	#n1,A	29	n1			x	x	x	x	6	A := n1 + A + C
	#n1,B	59	n1			x	x	x	x	6	B := n1 + B + C
ADD	B,A	68				x	x	x	x	8	A := B + A
	Rn1,A	18	n1			x	x	x	x	7	A := Rn1 + A
	Rn1,B	38	n1			x	x	x	x	7	B := Rn1 + B
	Rn1,Rn2	48	n1	n2		x	x	x	x	9	Rn2 := Rn1 + Rn2
	#n1,Rn2	78	n1	n2		x	x	x	x	8	Rn2 := n1 + Rn2
	#n1,A	28	n1			x	x	x	x	6	A := n1 + A
	#n1,B	58	n1			x	x	x	x	6	B := n1 + B
AND	B,A	63				0	x	x	0	8	A := B AND A
	A,Pn1	83	n1			0	x	x	0	9	Pn1 := Pn1 AND A
	Rn1,A	13	n1			0	x	x	0	7	A := Rn1 AND A
	#n1,A	23	n1			0	x	x	0	6	A := n1 AND A
	B,Pn1	93	n1			0	x	x	0	9	Pn1 := Pn1 AND A
	Rn1,B	33	n1			0	x	x	0	7	B := Rn1 AND B
	#n1,B	53	n1			0	x	x	0	6	B := n1 AND B
	Rn1,Rn2	43	n1	n2		0	x	x	0	9	Rn2 := Rn1 AND Rn2
	#n1,Pn2	A3	n1	n2		0	x	x	0	10	Pn2 := Pn2 AND n2
	#n1,Rn2	73	n1	n2		0	x	x	0	8	Rn2 := n1 AND Rn2
BR	n1n2	8C	n1	n2		–	–	–	–	9	PC := n1:n2
	@Rn1	9C	n1			–	–	–	–	8	PC := Rn1-1:Rn1
	n1n2(B)	AC	n1	n2		–	–	–	–	11	PC := n1:n2 + B (B = 8-bit unsigned index)
	n1(Rn2) (See Note 1)	F4 EC	n1	n2		–	–	–	–	16	PC := n1 + Rn2-1:Rn2 (n1 = 8-bit signed offset)
BTJO	B,A,n1	66	n1			0	x	x	0	10/12	If B AND A ≠0, PC := UPC + n1
	#n1,A,n2	26	n1	n2		0	x	x	0	8/10	If n1 AND A ≠ 0, PC := UPC + n2
	A,Pn1,n2	86	n1	n2		0	x	x	0	10/12	If A AND Pn1 ≠ 0, PC := UPC + n2
	Rn1,A,n2	16	n1	n2		0	x	x	0	9/11	If Rn1 AND A ≠ 0, PC := UPC + n2
	#n1,B,n2	56	n1	n2		0	x	x	0	8/10	If n1 AND B ≠ 0, PC := UPC + n2
	B,Pn1,n2	96	n1	n2		0	x	x	0	10/12	If B AND Pn1 ≠ 0, PC := UPC + n2
	Rn1,B,n2	36	n1	n2		0	x	x	0	9/11	If Rn1 AND B ≠ 0, PC := UPC + n2
	Rn1,Rn2,n3	46	n1	n2	n3	0	x	x	0	11/13	If Rn1 AND Rn2 ≠ 0, PC := UPC + n3
	#n1,Pn2,n3	A6	n1	n2	n3	0	x	x	0	11/13	If n1 AND Pn2 ≠ 0, PC := UPC + n3
	#n1,Rn2,n3 (See Notes 6,8 for all)	76	n1	n2	n3	0	x	x	0	10/12	If n1 AND Rn2 ≠ 0, PC := UPC + n3

† x = don't care, – = not applicable

Instruction Format		Opcode	Operands			Status Flags†				Cycles	Action Description
		1 2	1	2	3	C	N	Z	V		
BTJZ	B,A,n1	67	n1			0	x	x	0	10/12	If B AND NOT A ≠ 0, PC := UPC + n1
	#n1,A,n2	27	n1	n2		0	x	x	0	8/10	If n1 AND NOT A ≠ 0, PC := UPC + n2
	A,Pn1,n2	87	n1	n2		0	x	x	0	10/12	If A AND NOT Pn1 ≠ 0, PC := UPC + n2
	Rn1,A,n2	17	n1	n2		0	x	x	0	9/11	If Rn1 AND NOT A ≠ 0, PC := UPC + n2
	#n1,B,n2	57	n1	n2		0	x	x	0	8/10	If n1 AND NOT B ≠ 0, PC := UPC + n2
	B,Pn1,n2	97	n1	n2		0	x	x	0	10/12	If B AND NOT Pn1 ≠ 0, PC := UPC + n2
	Rn1,B,n2	37	n1	n2		0	x	x	0	9/11	If Rn1 AND NOT B ≠ 0, PC := UPC + n2
	Rn1,Rn2,n3	47	n1	n2	n3	0	x	x	0	11/13	If Rn1 AND NOT Rn2 ≠ 0, PC := UPC + n3
	#n1,Pn2,n3	A7	n1	n2	n3	0	x	x	0	11/13	If n1 AND NOT Pn2 ≠ 0, PC := UPC + n3
	#n1,Rn2,n3	77	n1	n2	n3	0	x	x	0	10/12	If n1 AND NOT Rn2 ≠ 0, PC := UPC + n3
(See Notes 6,8 for all)											
CALL	n1n2	8E	n1	n2		–	–	–	–	13	Push PC; PC := n1:n2
	@Rn1	9E	n1			–	–	–	–	12	Push PC; PC := Rn1-1:Rn1
	n1n2(B)	AE	n1	n2		–	–	–	–	15	Push PC; PC := n1:n2 + B (B = 8-bit unsigned index)
	n1(Rn2) (see Note 1)	F4 EE	n1	n2		–	–	–	–	20	Push PC; PC := n1 + Rn2-1:Rn2 (n1 = 8-bit signed offset)
CALLR	n1n2	8F	n1	n2		–	–	–	–	15	Push PC; PC := n1:n2 + UPC
	@Rn1	9F	n1			–	–	–	–	14	Push PC; PC := Rn1-1:Rn1 + UPC
	n1n2(B)	AF	n1	n2		–	–	–	–	17	Push PC; PC := n1:n2 + B + UPC (B = 8-bit unsigned index)
	n1(Rn2)	F4 EF	n1	n2		–	–	–	–	22	Push PC; PC := n1 + Rn2-1:Rn2 + UPC (n1 = 8-bit signed offset)
(See Note 1 for all)											
CLR	A	B5				0	0	1	0	8	A := 00
	B	C5				0	0	1	0	8	B := 00
	Rn1	D5	n1			0	0	1	0	8	Rn1 := 00
CLRC		B0				0	x	x	0	9	CNZV := 0xx0
CMP	B,A	6D				x	x	x	x	8	A - B
	#n1,A	2D	n1			x	x	x	x	6	A - n1
	Rn1,A	1D	n1			x	x	x	x	7	A - Rn1
	n1n2,A	8D	n1	n2		x	x	x	x	11	A - @(n1:n2)
	@Rn1,A	9D	n1			x	x	x	x	10	A - @(Rn1-1:Rn1)
	n1(SP),A (See Note 1)	F3	n1			x	x	x	x	8	A - @(SP) + n1) (n1 = 8-bit signed offset)
	n1n2(B),A	AD	n1	n2		x	x	x	x	13	A - @(n1:n2 + B) (B = 8-bit unsigned index)
	n1(Rn2),A (See Note 1)	F4 ED	n1	n2		x	x	x	x	18	A - @(n1 + Rn2-1l:Rn2) (n1 = 8-bit signed offset)
	#n1,B	5D	n1			x	x	x	x	6	B - n1
	Rn1,B	3D	n1			x	x	x	x	7	B - Rn1
	Rn1,Rn2	4D	n1	n2		x	x	x	x	9	Rn2 - Rn1
	#n1,Rn2	7D	n1	n2		x	x	x	x	8	Rn2 - n1

† x = don't care, – = not applicable

Instruction Format	Opcode	Operands			Status Flags†	Cycles	Action Description
	1 2	1	2	3	C N Z V		
COMPL A	BB				x x x 0	8	A := 100h - A ‡
B	CB				x x x 0	8	B := 100h - B ‡
Rn1	DB	n1			x x x 0	10	Rn1 := 100h - Rn1 ‡
(See Note 1 for all)							
DAC B,A	6E				x x x x	10	A := B + A + C §
Rn1,A	1E	n1			x x x x	9	A := Rn1 + A + C §
Rn1,B	3E	n1			x x x x	9	B := Rn1 + B + C §
Rn1,Rn2	4E	n1	n2		x x x x	11	Rn2 := Rn1 + C §
#n1,Rn2	7E	n1	n2		x x x x	10	Rn2 := n1 + Rn2 + C §
#n1,A	2E	n1			x x x x	8	A := n1 + A + C §
#n1,B	5E	n1			x x x x	8	B := n1 + B + C §
DEC A	B2				x x x x	8	A := A -1
B	C2				x x x x	8	B := B - 1
Rn1	D2	n1			x x x x	6	Rn1 := Rn1 - 1
DINT (See Note 2)	F0 00				0 0 0 0	6	ST := 00
DIV Rn1,A	F4 F8	n1			0 x x 0 1 1 1 1	47-63 14	A,B := A:B / Rn1 If no overflow else: A, B, Rn1 unaffected. A = quotient; B = remainder
(See Notes 1,3)							
DJNZ A,n1	BA	n1			– – – –	10/12	A := A - 1; If A ≠ 0 then PC := PC + n1 + 2
B,n1	CA	n1			– – – –	10/12	B := B - 1; If B ≠ 0 then PC := PC + n1 + 2
Rn1,n2	DA	n1	n2		– – – –	8/10	Rn1 - 1; If Rn1 ≠ 0 then PC := PC + n2 + 3
(See Note 8 for all)							(n1 = 8-bit signed offset)
DSB B,A	6F				x x x x	10	A := A - B - C §
Rn1,A	1F	n1			x x x x	9	A := A - Rn1 - C §
Rn1,B	3F	n1			x x x x	9	B := B - Rn1 - C §
Rn1,Rn2	4F	n1	n2		x x x x	11	Rn2 := Rn2 - Rn1 - C §
#n1,Rn2	7F	n1	n2		x x x x	10	Rn2 := Rn2 - n1 - C §
#n1,A	2F	n1			x x x x	8	A := A - n1 - C §
#n1,B	5F	n1			x x x x	8	B := B - n1 - C §
EINT (See Note 2)	F0 0C				0 0 0 0	6	ST := 0C
EINTH (See Note 2)	F0 04				0 0 0 0	6	ST := 04
EINTL (See Notes 1,2)	F0 08				0 0 0 0	6	ST := 08
IDLE	F6				– – – –	6	stop instruction execution until interrupt
(See Note 5)							
INC A	B3				x x x x	8	A := A + 1
B	C3				x x x x	8	B := B + 1
Rn1	D3	n1			x x x x	6	Rn1 := Rn1 +1

† x = don't care, – = not applicable
‡ Form 2's complement
§ Operands are in BCD

Instruction Format		Opcode	Operands			Status Flags†				Cycles	Action Description
		1 2	1	2	3	C	N	Z	V		
INCW	#n1,Rn2 (See Notes 1,4)	70	n1	n2		x	x	x	x	11	Rn2-1:Rn2 := Rn2-1:Rn2 + n1 (signed 2's complement, n1 = 8-bit signed offset)
INV	A	B4				0	x	x	0	8	A := NOT A
	B	C4				0	x	x	0	8	B := NOT B
	Rn1	D4	n1			0	x	x	0	6	Rn1 := NOT Rn1
JMP	n1 (See Note 6)	00	n1			–	–	–	–	7	PC := n1 + UPC ‡
JMPL	n1n2	89	n1	n2		–	–	–	–	9	PC := n1:n2 + UPC ‡
	@Rn1	99	n1			–	–	–	–	8	PC := Rn1-1:Rn1 + UPC ‡
	n1n2(B)	A9	n1	n2		–	–	–	–	11	PC := n1:n2 + B + UPC ‡ (B = 8-bit unsigned index)
	n1(Rn2) (See Notes 1,7 for all)	F4 E9	n1	n2		–	–	–	–	16	PC := n1 + Rn2-1:Rn2 + UPC ‡ (n1 = 8-bit signed offset)
JN	n1 (Note 6,8)	01	n1			–	–	–	–	5/7	If N = 1, PC := UPC + n1
JZ/JEQ	n1 (Note 6,8)	02	n1			–	–	–	–	5/7	If Z = 1, PC := UPC + n1
JC	n1 (Note 6,8)	03	n1			–	–	–	–	5/7	If C = 1, PC := UPC + n1
JP	n1 (Note 6,8)	04	n1			–	–	–	–	5/7	If N = 0 AND Z = 0, PC := UPC + n1
JPZ	n1 (Note 6,8)	05	n1			–	–	–	–	5/7	If N = 0, PC := UPC + n1
JNZ JNE	n1 (Note 6,8) (Note 6,8)	06	n1			–	–	–	–	5/7	If Z = 0, PC := UPC + n1
JNC	n1 (Note 6,8)	07	n1			–	–	–	–	5/7	If C = 0, PC := UPC + n1
JV	n1 (Note 1,6,8)	08	n1			–	–	–	–	5/7	If V = 0, PC := UPC + n1
JNV	n1 (Note 1,6,8)	0C	n1			–	–	–	–	5/7	If V = 0, PC := UPC + n1
JGE	n1 (Note 1,6,8)	0D	n1			–	–	–	–	5/7	If N XOR V = 0, PC := UPC + n1
JL	n1 (Note 1,6,8)	09	n1			–	–	–	–	5/7	If N XOR V = 1, PC := UPC + n1
JG	n1 (Note 1,6,8)	0E	n1			–	–	–	–	5/7	If Z OR (N XOR V) = 0, PC := UPC + n1
JLE	n1 (Note 1,6,8)	0A	n1			–	–	–	–	5/7	If Z OR (N XOR V) = 1, PC := UPC + n1
JLO	n1 (Note 1,6,8)	0F	n1			–	–	–	–	5/7	If C = 0 AND Z = 0, PC := UPC + n1
JHS	n1 (Note 1,6,8)	0B	n1			–	–	–	–	5/7	If C = 1 OR Z = 1,PC := UPC + n1 (n1 = 8-bit signed offset)
LDSP		FD				–	–	–	–	7	SP := B
LDST	#n1	F0	n1			x	x	x	x	6	ST := n1

† x = don't care, – = not applicable
‡ Unconditional

Instruction Format		Opcode		Operands			Status Flags†				Cycles	Action Description
		1	2	1	2	3	C	N	Z	V		
MOV	A,B	C0					0	x	x	0	9	B := A
	A,Pn1	21		n1			0	x	x	0	8	Pn1 := A
	A,Rn1	D0		n1			0	x	x	0	7	Rn1 := A
	A,@Rn1	9B		n1			0	x	x	0	9	@(Rn1-1:Rn1) := A
	A,n1(SP)	F2		n1			0	x	x	0	7	@(SP) + n1 := A
	(See Note 1)											(n1 = 8-bit signed offset)
	A,n1n2	8B		n1	n2		0	x	x	0	10	@(n1:n2) := A
	A,n1n2(B)	AB		n1	n2		0	x	x	0	12	@(B + n1:n2) := A
												(B has 8-bit unsigned index)
	A,n1(Rn2)	F4	EB	n1	n2		0	x	x	0	17	@(n1 + Rn2-1:Rn2) := A
	(See Note 1)											(n1 = 8-bit signed offset)
	#n1,A	22		n1			0	x	x	0	6	A := n1
	Pn1,A	80		n1			0	x	x	0	8	A := Pn1
	Pn1,Rn2	A2		n2	n1		0	x	x	0	10	Rn2 := Pn1 (reversed operand order)
	Rn1,A	12		n1			0	x	x	0	7	A := Rn1
	Rn1,Pn2	71		n2	n1		0	x	x	0	10	Pn2 := Rn1 (reversed operand order)
	@Rn1,A	9A		n1			0	x	x	0	9	A := @(Rn1-1:Rn1)
	n1n2,A	8A		n1	n2		0	x	x	0	10	A := @(n1:n2)
	n1n2(B),A	AA		n1	n2		0	x	x	0	12	A := @(B + n1:n2)
												(B has an 8-bit unsigned index)
	n1(Rn2),A	F4	EA	n1	n2		0	x	x	0	17	A := @(n1 + Rn2-1:Rn2)
	(See Note 1)											(n1 = 8-bit signed offset)
	n1(SP),A	F1		n1			0	x	x	0	7	A := @(SP + n1)
	(See Note 1)											(n1 = 8-bit signed offset)
	B,A	62					0	x	x	0	8	A := B
	B,Pn1	51		n1			0	x	x	0	8	Pn1 := B
	B,Rn1	D1		n1			0	x	x	0	7	Rn1 := B
	#n1,B	52		n1			0	x	x	0	6	B := n1
	Pn1,B	91		n1			0	x	x	0	8	B := B
	Rn1,B	32		n1			0	x	x	0	7	B := Rn1
	Rn1,Rn2	42		n1	n2		0	x	x	0	9	Rn2 := Rn1
	#n1,Rn2	72		n1	n2		0	x	x	0	8	Rn2 := n1
	#n1,Pn2	F7		n1	n2		0	x	x	0	10	Pn2 := n1
MOVW	#n1n2,Rn3	88		n1	n2	n3	0	x	x	0	13	Rn3-1:Rn3 := n1:n2
	Rn1,Rn2	98		n1	n2		0	x	x	0	12	Rn2-1:Rn2 :=Rn1-1:Rn1
	#n1(Rn2),Rn3	F4	E8	n1	n2	n3	0	x	x	0	20	Rn3-1:Rn3 := n1 + Rn2-1:Rn2
	(See Note 1)											(n1 = 8-bit signed offset)
	#n1n2(B),Rn3	A8		n1	n2	n3	0	x	x	0	15	Rn3-1:Rn3 := n1:n2 + B
MPY	B,A	6C					0	x	x	0	47	A:B := B × A
	Rn1,A	1C		n1			0	x	x	0	46	A:B := Rn1 × A
	Rn1,B	3C		n1			0	x	x	0	46	A:B := Rn1 × B
	Rn1,Rn2	4C		n1	n2		0	x	x	0	48	A:B := Rn1 × Rn2
	#n1,Rn2	7C		n1	n2		0	x	x	0	47	A:B := n1 × Rn2
	#n1,A	2C		n1			0	x	x	0	45	A:B := n1 × A
	#n1,B	5C		n1			0	x	x	0	45	A:B := n1 × B
NOP		FF					–	–	–	–	7	PC := PC + 1

† x = don't care, – = not applicable

Instruction Format		Opcode	Operands			Status Flags†				Cycles	Action Description
		1 2	1	2	3	C	N	Z	V		
OR	B,A	64				0	x	x	0	8	A := B OR A
	A,Pn1	84	n1			0	x	x	0	9	Pn1 := Pn1 OR A
	Rn1,A	14	n1			0	x	x	0	7	A := Rn1 OR A
	#n1,A	24	n1			0	x	x	0	6	A := n1 OR A
	B,Pn1	94	n1			0	x	x	0	9	Pn1 := Pn1 OR B
	Rn1,B	34	n1			0	x	x	0	7	B := Rn1 OR B
	#n1,B	54	n1			0	x	x	0	6	B := n1 OR B
	Rn1,Rn2	44	n1	n2		0	x	x	0	9	Rn2 := Rn1 OR Rn2
	#n1,Pn2	A4	n1	n2		0	x	x	0	10	Pn2 := Pn1 OR n2
	#n1,Rn2	74	n1	n2		0	x	x	0	8	Rn2 := n1 OR Rn2
POP	A	B9				0	x	x	0	9	A := @SP; SP := SP - 1
	B	C9				0	x	x	0	9	B := @SP; SP := SP - 1
	Rn1	D9	n1			0	x	x	0	7	Rn1 := @SP; SP := SP - 1
	ST	FC				0	x	x	0	8	ST := @SP; SP := SP -1
PUSH	A	B8				0	x	x	0	9	SP := SP + 1; @SP := A
	B	C8				0	x	x	0	9	SP := SP + 1; @SP := B
	Rn1	D8	n1			0	x	x	0	7	SP := SP + 1; @SP := Rn1
	ST	FB	n1			–	–	–	–	8	SP := SP + 1; @SP := ST
RL	A	BE				x	x	x	0	8	C := A(7); A := A(6,5,4,3,2,1,0,7)
	B	CE				x	x	x	0	8	C := B(7); B := B(6,5,4,3,2,1,0,7)
	Rn1	DE	n1			x	x	x	0	6	C := Rn1(7); Rn1 := B(6,5,4,3,2,1,0,7)
RLC	A	BF				x	x	x	0	8	C := A(7); A := A(6,5,4,3,2,1,0,C)
	B	CF				x	x	x	0	8	C := B(7); B := B(6,5,4,3,2,1,0,C)
	Rn1	DF	n1			x	x	x	0	6	C := Rn1(7); Rn1 : = Rn1(6,5,4,3,2,1,0,C)
RR	A	BC				x	x	x	0	8	C := A(0); A := A(0,7,6,5,4,3,2,1)
	B	CC				x	x	x	0	8	C := B(0); B := B(0,7,6,5,4,3,2,1)
	Rn1	DC	n1			x	x	x	0	6	C := Rn1(0); Rn1 := Rn1(0,7,6,5,4,3,2,1)
RRC	A	BD				x	x	x	0	8	C := A(0); A := A(C,7,6,5,4,3,2,1)
	B	CD				x	x	x	0	8	C := B(0); B := B(C,7,6,5,4,3,2,1)
	Rn1	DD				x	x	x	0	6	C := Rn1(0); Rn1 := Rn1(C,7,6,5,4,3,2,1)
RTI		FA				x	x	x	x	12	Pop PCl; Pop PCh; Pop ST
RTS		F9				–	–	–	–	9	Pop PCl; Pop PCh
SETC		F8				1	0	1	0	7	CNZV := 1010
SBB	B,A	6B				x	x	x	x	8	A := A - B - C
	Rn1,A	1B	n1			x	x	x	x	7	A := A - Rn1 - C
	#n1,A	2B	n1			x	x	x	x	6	A := A - n1 - C
	Rn1,B	3B	n1			x	x	x	x	7	B := B - Rn1 - C
	#n1,B	5B	n1			x	x	x	x	6	B := B - n1 - C
	Rn1,Rn2	4B	n1	n2		x	x	x	x	9	Rn2 := Rn2 - Rn1 - C
	#n1,Rn2	7B	n1	n2		x	x	x	x	8	Rn2 := Rn2 - n1 - C

† x = don't care, – = not applicable

Instruction Format		Opcode	Operands			Status Flags†				Cycles	Action Description
		1 2	1	2	3	C	N	Z	V		
STSP		FE				–	–	–	–	8	B := SP
SUB	B,A	6A				x	x	x	x	8	A := A - B
	Rn1,A	1A	n1			x	x	x	x	7	A := A - Rn1
	#n1,A	2A	n1			x	x	x	x	6	A := A - n1
	Rn1,B	3A	n1			x	x	x	x	7	B := B - Rn1
	#n1,B	5A	n1			x	x	x	x	6	B := B - n1
	Rn1,Rn2	4A	n1	n2		x	x	x	x	9	Rn2 := Rn2 - Rn1
	#n1,Rn2	7A	n1	n2		x	x	x	x	8	Rn2 := Rn2 - n1
SWAP	A	B7				0	x	x	0	11	A(7:4,3:0) := A(3:0,7:4)
	B	C7				0	x	x	0	11	B(7:4,3:0) := B(3:0,7:4)
	Rn1	D7	n1			0	x	x	0	9	Rn1(7:4,3:0) := Rn1(3:0,7:4)
TRAP	0	EF				–	–	–	–	14	Push PC; PCh := @(7FDEh); PCl := @(7FDFh)
	1	EE				–	–	–	–	14	Push PC; PCh := @(7FDCh); PCl := @(7FDDh)
	2	ED				–	–	–	–	14	Push PC; PCh := @(7FDAh); PCl := @(7FDBh)
	3	EC				–	–	–	–	14	Push PC; PCh := @(7FD8h); PCl := @(7FD9h)
	4	EB				–	–	–	–	14	Push PC; PCh := @(7FD6h); PCl := @(7FD7h)
	5	EA				–	–	–	–	14	Push PC; PCh := @(7FD4h); PCl := @(7FD5h)
	6	E9				–	–	–	–	14	Push PC; PCh := @(7FD2h); PCl := @(7FD3h)
	7	E8				–	–	–	–	14	Push PC; PCh := @(7FD0h); PCl := @(7FD1h)
	8	E7				–	–	–	–	14	Push PC; PCh := @(7FCEh); PCl := @(7FCFh)
	9	E6				–	–	–	–	14	Push PC; PCh := @(7FCCh); PCl := @(7FCDh)
	10	E5				–	–	–	–	14	Push PC; PCh := @(7FCAh); PCl := @(7FCBh)
	11	E4				–	–	–	–	14	Push PC; PCh := @(7FC8h); PCl := @(7FC9h)
	12	E3				–	–	–	–	14	Push PC; PCh := @(7FC6h); PCl := @(7FC7h)
	13	E2				–	–	–	–	14	Push PC; PCh := @(7FC4h); PCl := @(7FC5h)
	14	E1				–	–	–	–	14	Push PC; PCh := @(7FC2h); PCl := @(7FC3h)
	15	E0				–	–	–	–	14	Push PC; PCh := @(7FC0h); PCl := @(7FC1h)

† x = don't care, – = not applicable

Instruction Format		Opcode		Operands			Status Flags†				Cycles	Action Description
		1	2	1	2	3	C	N	Z	V		
TST	A	B0					0	x	x	0	9	A - 0
	B	C6					0	x	x	0	10	B - 0
XCHB	A	B6					0	x	x	0	10	A := B; B := A
	B	C6					0	x	x	0	10	B := B
	Rn1	D6		n1			0	x	x	0	8	Rn1 := B; B := Rn1
XOR	B,A	65					0	x	x	0	8	A := B XOR A
	A,Pn1	85		n1			0	x	x	0	9	Pn1 := Pn1 XOR A
	Rn1,A	15		n1			0	x	x	0	7	A := Rn1 XOR A
	#n1,A	25		n1			0	x	x	0	6	A := n1 XOR A
	B,Pn1	95		n1			0	x	x	0	9	Pn1 := Pn1 XOR B
	Rn1,B	35		n1			0	x	x	0	7	B := Rn1 XOR B
	#n1,B	55		n1			0	x	x	0	6	B := n1 XOR B
	Rn1,Rn2	45		n1	n2		0	x	x	0	9	Rn2 := Rn1 XOR Rn2
	#n1,Pn2	A5		n1	n2		0	x	x	0	10	Pn2 := Pn1 XOR n2
	#n1,Rn2	75		n1	n2		0	x	x	0	8	Rn2 := n1 XOR Rn2

† x = don't care, – = not applicable

Note that **PC** denotes the address of the current instruction. The value used at execution time for program-counter-relative operand and branch addresses is the UPC. Thus, the symbolic instruction `JMP $` (where `$` is the address of the instruction) has an object code of 00 FEh. This effectively subtracts 2 from the contents of the UPC and causes a refetch of the current instruction.

Notes:

1. No equivalent instruction in TMS7000 processors.
2. These instructions transfer the second byte of the opcode into the Status Register.
3. A 16-bit dividend (A:B) divided by an 8-bit divisor (Rn1) yields an 8-bit quotient (A) and an 8-bit remainder (B), all unsigned. Execution time is related to the number of ones in the quotient with a base level of 47 cycles and a maximum of 63 cycles. Overflow conditions are checked explicitly prior to actual division; if detected, the operands are left unchanged, the C, N, Z, and V bits of the status register are all set and the instruction is aborted after 14 cycles.
4. INCW algebraically adds the signed 8-bit immediate operand to the content of the specified register pair.
5. Stops execution and opcode fetches; activates system STANDBY or HALT mode based on content of Configuration Control Registers.
6. Operand generates 8-bit signed offset to Program Counter.
7. Operand generates 16-bit signed offset to Program Counter.
8. Execution cycles of conditional jumps is the lesser of the two values given if the jump is not taken.

7. Macro Language

The assembler supports a macro language that allows you to create your own "commands." This is especially useful when a program executes a particular task several times.

The macro language allows you to:

- Define your own macros
- Redefine existing opcodes and macros
- Access macro libraries created with the archiver
- Manipulate strings within a macro
- Define conditional and repeatable blocks within a macro
- Control macro expansion listing

There are three phases of macro use:

- **Macro definition.** Macros must be defined before they can be invoked. There are two methods for defining macros:

  1) Macros can be defined in the **source file** where they are used (or in a separate text file that is included with a .include directive). Since macros must be defined before they are called, it is a good practice to place all the definitions at the beginning of the file.

  2) 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 that corresponds to the name of the member. You can access a macro library by using the .mlib directive. Since macros must be defined before they can be called, the .mlib directive must appear in the source code before any of the macros in the library are called.

- **Macro invocation.** Once a macro has been defined, the macro name can be used as an opcode in a source program. This is referred to as a *macro call.*
- **Macro expansion.** When the source program calls a macro, the assembler substitutes the statements within the macro definition for the macro call statement.

This section discusses the following topics:

Section **Page**

7.1 Macro Directives Summary

Directive	Description
.MACRO	*Macro Definition Directive* **Syntax:**<macro name> .MACRO [<parm 1>][,...,<parm n>] The .MACRO directive begins a macro definition. It must be the first statement in a macro definition. .MACRO assigns a name to the macro and declares the macro parameters. *Macro name* is the name of the macro. A macro name may be 1 to 32 alphanumeric characters; it must begin with a letter. *Parms* are optional parameters. When a macro is called, the assembler will associate the first operand in the macro call with the first parameter of the macro definition, and so on.
.VAR	*Variable Declaration Directive* **Syntax:** .VAR <var 1>[,...,<var n>] The .VAR directive declares variables that can be used within a macro definition. .VAR is only necessary for declaring variables that are not parameters. Up to 128 variables can be declared within one macro. You can use more than one .VAR statement per macro; each .VAR statement may declare several variables. Only the first 8 characters of a variable name are significant.
.ASG	*Assign Value to Variable Directive* **Syntax:** .ASG <expression or string> , <var> The .ASG directive assigns values to variables that have been declared with the .VAR directive or passed as parameters.
.IF	*Begin Conditional Block Directive* **Syntax:** .IF <expression> The .IF directive begins a conditional block. If the *expression* evaluates to a non-zero value, then the code following the .IF directive (up to an .ELSE or .ENDIF directive) will be assembled.
.ELSE	*Alternate Conditional Block Directive* **Syntax:** .ELSE The .ELSE directive may be used within a conditional block. If the *expression* in an .IF directive evaluates to 0, then code following a corresponding .ELSE directive (up to an .ENDIF directive) will be assembled.
.ENDIF	*Terminate Conditional Block Directive* **Syntax:** .ENDIF The .ENDIF directive terminates a conditional block.
.ENDM	*Terminate Macro Definition Directive* **Syntax:** .ENDM The .ENDM directive terminates a macro definition.
.LOOP	*Begin Repeatable Block Directive* **Syntax:** .LOOP <expression> The .LOOP directive begins a repeatable block. The expression is evaluated only once; the expression should evaluate to a value in the range 0–32767.
.ENDLOOP	*Terminate Repeatable Block Directive* **Syntax:** .ENDLOOP The .ENDLOOP directive terminates a repeatable block.

7.2 Macro Libraries

A macro library is a collection of files that contain macro definitions. These files,or members, are bound into a single file (called an archive) by the archiver. Each member of a macro library may contain one macro definition; the macro name and the member name must be the same. 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 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 only extracted 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 following example creates a macro library called `maclib.lib`:

```
370ar -a maclib.lib mac1.asm mac2.asm mac3.asm mac4.asm
```

This example adds four macro files (`mac1.asm`, `mac2.asm`, `mac3.asm`, and `mac4.asm`) to the library `maclib.lib`. Note that this could be a new or an existing library; if the library already existed, this example would simply append the macros to the end of the library.

Now you can specify `maclib.lib` to the assembler with an .mlib directive, and call any of the macros that it contains:

```
.mlib    "maclib.lib"
mac1                    ; Macro call
```

The assembler assumes that the files in the archive contain macro definitions with the same names as the members. The assembler expects **only** macro definitions in a macro library; putting object code or miscellaneous source files into the library may produce undesirable effects.

7.3 Defining Macros

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

```
<macname>   .MACRO [<parm1>] [,<parm2>] ... [,<parmn>]
            .
            .
            .
            <model statements>
            or
            <macro directives>
            .
            .
            .
            .ENDM
```

where:

macname names the macro. It must be placed in the source statement's label field. Macro names are significant to 32 characters. The assembler places this name in the internal opcode table, replacing any instruction or previous macro definition with the same name.

.MACRO identifies this source statement as the first line of a macro definition; it must be placed in the opcode field.

parms are optional parameters which may appear as operands for the .MACRO directive. Parameters are not required by all macros.

model statements
are instructions or assembler directives that are used each time the macro is invoked.

macro directives
control the expansion of the macro or manipulate macro variables.

.ENDM terminates the macro definition.

The contents of a single macro definition must be contained in the same file. Macro definitions cannot be nested, but other directives, instructions, and macro calls may be used in a macro definition. The assembler performs only limited error checking of macro definitions (during the definition phase), so multiple expansions of a macro may produce duplicate error messages.

When a macro is called, the assembler will substitute the model statements and macro directives within the definition for the macro call in the source code. Example 7-1 shows an example of a macro definition, how it could be called, and how it would be expanded in the source code.

Example 7-1. Macro Definition and Expansion

Macro Definition: The following code defines a macro, MOVREG, that has three parameters.

```
0000                0001 ;***********************************************
0000                0002 MOVREG    .MACRO    p1,p2,pN
0000                0003           mov       :p1:,:p2:
0000                0004           mov       :p2:,:pN:
0000                0005           .LOOP     2
0000                0006           nop
0000                0007           .ENDLOOP
0000                0008           .ENDM
0000                0009 ;***********************************************
```

Macro Call: The MOVREG macro is invoked in the source code.

```
0000                0010 ;***********************************************
0000                0011
0000                0012           MOVREG    R0,R1,R5
```

Macro Expansion: The assembler substitutes the functional lines of the macro definition for the macro call. The macro parameters are replaced with the operands supplied in the macro call.

```
0000 420001         0001 #             mov       R0,R1
0003 420105         0002 #             mov       R1,R5
0006 FF             0003 #             nop
0007 FF             0004 #             nop
```

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 to add new instructions.

Caution:

When you specify a macro library with the .mlib directive, the assembler places all the entries in the specified library into the opcode table - not just the macros that are called. Make sure that the macros and instructions you want to use are not redefined by macros in a macro library.

7.4 Macro Variables

Macros can declare local variables whose scope is limited to the defining macro. These variables do not conflict with symbols defined outside the macro. Only the first eight characters of a variable name are significant. A single macro can declare a maximum of 128 variables.

A variable can be defined in one of two ways:

- As a parameter defined by the .MACRO directive. The assembler assigns initial values to macro parameters when the macro is called.

  As an example, consider the following macro definition line:

  ```
  ADDUP   .MACRO    val1,val2,sum
  ```

  This example defines three variables (`val1`, `val2`, and `sum`). The assembler assigns values to these variables when it expands the macro; each parameter corresponds to an operand in the macro call.

- As a local variable that appears as the operand of a .VAR directive. These variables have initial values of 0; you can assign values to them with the .ASG directive.

  The following macro line:

  ```
          .VAR      color1,border
  ```

  defines two local variables, `color1` and `border`, which have initial values of 0.

The macro language provides a string substitution facility that allows you to build instructions out of strings that are stored in the variables. These string values can be assigned, operated on, concatenated, and substituted into model statements.

7.4.1 Variable Values

Values are assigned to:

- *Parameters* during a macro call.
- *Local variables* with the .ASG directive.

The .ASG directive assigns a value to a variable. The syntax of the .ASG directive is:

```
.ASG    <macro expression> , <macro variable[.component]>
```

The macro expression can contain a string, a constant, or an expression. The main use of .ASG is to assign values to local macro variables which have no useful value; it can also be used to put values in the assembler symbol table.

Example 7-2 shows a macro that has two variables. The variable `defrost` is a parameter; the variable `temp_control` is a local variable.

Example 7-2. Assigning Values to Variables

```
0000                    0001 climate    .MACRO    defrost
0000                    0002            .VAR      temp_control
0000                    0003            .ASG      3 , temp_control
0000                    0004            .byte     :temp_control:
0000                    0005            .byte     :defrost:
0000                    0006            .ENDM
0000                    0007
004E                    0008 setting    .equ      4Eh
0000                    0009            climate   setting
0000 03                 0001 #             .byte     3
0001 4E                 0002 #             .byte     setting
```

In this example, the variable `defrost` automatically has the same value as the constant `setting`, because `setting` was passed as a parameter. The variable `temp_control` has a value that was assigned to it with an .ASG directive.

The value that is assigned to a macro variable is called a **string value**. The assembler will substitute a variable's string value into a model statement when you enclose the variable in colons. Variables can be used this way anywhere in a model statement (as a label, an operand, etc.). Model statements are only scanned once, and expanded strings are not rescanned by the assembler. Note that qualified macro variables can also appear between colons.

7.4.2 Qualifying Variables

In *addition* to the value assigned through macro calls or .ASG statements (the string value), each macro variable has up to seven *components*. A variable component provides additional information about the variable. It is accessed through the use of 1- or 2-letter suffixes in the following format:

name.suffix

There are two types of components:

- Macro components, which provide information about macro variables.
- Symbol components, which provide information about symbols that are defined in the assembly language program.

Any variable component can be used in an expression and assigned to a component of a variable or to a component of a symbol. Components of one variable can be assigned to components of another variable. If you assign a value to one component of a variable, only that component is assigned a value.

7.4.2.1 Macro Components

Macro components provide information about macro variables. The components of parameters are set when the macro is invoked. The components of local variables are initially set to 0. Table 7-1 lists the four macro variable components.

Table 7-1. Macro Components

Component	Description
S	*String component.* This is the default component when no suffix is provided. It is equivalent to :variable:. For a macro parameter, the string component is a copy of the string that passes to variable.
V	*Value component.* This component contains the variable's value. If the string represents a number or an expression, this component contains a binary equivalent of the value.
L	*Length component.* This component contains the number of characters that make up the string component.
A	*Attribute component.* This component contains information about the variable, such as: Was the parameter passed to the macro? 0 or missing parameters are legal.) Was it an operand list? Was it of a particular format (register, indirect, indexed, symbolic, etc.)?

Example 7-3 illustrates the use of macro components. In this example, `sym` is a constant that is passed to the macro `mac`. The variable `parm` is defined as a parameter; local variables `var1` and `var2` are defined with the .VAR directive.

- Lines 1–4 of the macro expansion show how values are substituted for the following components of the variable `parm`:

  `:parm:` specifies the string component of parameter `parm`, which is *sym*.
  `parm.s` also specifies the string component of `parm`.
  `parm.v` specifies the value component of `parm`, which is *5*.
  `parm.l` specifies the length component of `parm`, which is *3* (there are three characters in the string component).
  `parm.a` specifies the attribute component of `parm`; if used, it would indicate that `parm` was passed to the macro as a parameter.

- Lines 5–7 of the macro expansion show that you can enclose a variable and a component in colons. This causes the assembler to use a string representation of the string or value; this is especially useful for concatenating strings and values.

- Lines 8–10 of the macro expansion show the values of the symbol `var1` and its components. The string component of `parm` is assigned to the string component of `var1` with an .ASG directive. The string component of `var1` is *sym* and the length component of `var1` is *3*. However, the value component is 0, because no value was assigned to it.

- Lines 11–13 of the macro expansion show the values of the symbol `var2` and its components. The value component of `parm` is assigned to the value component of `var2` with an .ASG directive. The value component of `var2` is *5*. However, the string component is empty, because no string was assigned to it; the length component is 0, because there is no string component.

Example 7-3. Using Macro Components

```
   0000                 0001 mac       .MACRO    parm
   0000                 0002           .var      var1,var2
   0000                 0003
   0000                 0004           .word     :parm:
   0000                 0005
   0000                 0006           .word     parm.s
   0000                 0007           .word     parm.v
   0000                 0008           .word     parm.l
   0000                 0009
   0000                 0010           .word     :parm.s:
   0000                 0011           .word     :parm.v:
   0000                 0012           .word     :parm.l:
   0000                 0013
   0000                 0014           .ASG      parm.s , var1.s
   0000                 0015           .word     var1.s
   0000                 0016           .word     var1.v
   0000                 0017           .word     var1.l
   0000                 0018
   0000                 0019           .ASG      parm.v , var2.v
   0000                 0020           .word     var2.s
   0000                 0021           .word     var2.v
   0000                 0022           .word     var2.l
   0000                 0023
   0000                 0024           .ENDM
   0000                 0025
   0000                 0026 ;**************************************
   0000                 0027
   0005                 0028 sym       .equ      5
   0000                 0029           mac       sym
   0000 0005            0001 #            .word     sym
   0002 0005            0002 #            .word     sym
   0004 0005            0003 #            .word     5
   0006 0003            0004 #            .word     3
   0008 0005            0005 #            .word     sym
   000A 0005            0006 #            .word     5
   000C 0003            0007 #            .word     3
   000E 0005            0008 #            .word     sym
   0010 0000            0009 #            .word     0
   0012 0003            0010 #            .word     3
E  0014                 0011 #            .word
   0014 0005            0012 #            .word     5
   0016 0000            0013 #            .word     0
```

7.4.2.2 Symbol Components

All symbols that are defined in an assembly language program have four symbol components that are similar to the macro variable components described in Section 7.4.2.1. The only way you can access these symbol components is by assigning the symbol's name to macro variables in a macro. *To accomplish this, you must use an .ASG statement or pass the symbol name as a parameter to assign the symbol name to the string component of a macro variable.*

Table 7-2. Symbol Components

Component	Description
SS	*String component.* This is similar to the macro string component; it is the string name of the symbol. Initially, the string component is empty. It may contain any string assigned to it via an .ASG statement.
SV	*Value component.* This component contains the value of the symbol in the symbol table. This could be a relocatable address or a value which the symbol was equated to.
SL	*Length component.* This component contains the length of the string (if any) that has been assigned to the symbol string component of the symbol.
SA	*Attribute component.* This component contains information about the symbol, such as: Is the symbol relocatable? Is the symbol global? Has a string component been assigned to the symbol? Is the symbol defined? Is the symbol a macro name?

Example 7-4 illustrates the use of symbol components. This example contains three macros:

- `inc_const` has two parameters; the first should be a symbol, the second is a looping variable. This macro increments the symbol's value component in the symbol table the specified number of times.
- `s_bool` has two parameters; the first is a flag, and the second is a value (true or false). This macro sets the flag's string component in the symbol table to true or false.
- `q_bool` has one parameter, a flag. This macro prints the contents of the flag's string component in the symbol table by means of a .string directive.

Example 7-4. Using Symbol Components

```
0000                0001 ;************************************
0000                0002 ;**          macro inc_const          **
0000                0003 ;************************************
0000                0004 inc_const  .MACRO    p1,ntimes
0000                0005
0000                0006            .LOOP     ntimes.v
0000                0007            .ASG      p1.sv+1, p1.sv
0000                0008            .ENDLOOP
0000                0009
0000                0010            .ENDM
0000                0011
0000                0012 ;************************************
0000                0013 ;**          macro s_bool             **
0000                0014 ;************************************
0000                0015 s_bool     .MACRO    p,val
0000                0016
0000                0017            .ASG      val.s , p.ss
0000                0018
0000                0019            .ENDM
0000                0020
0000                0021 ;************************************
0000                0022 ;**          macro q_bool             **
0000                0023 ;************************************
0000                0024 q_bool     .MACRO    tfflag
0000                0025
0000                0026            .string   tfflag.ss
0000                0027
0000                0028            .ENDM
0000                0029
0000                0030 ;************************************
0000                0031 ;**              main                 **
0000                0032 ;************************************
0000                0033 inner      .equ      0
0005                0034 outer      .equ      5
0000                0035 iflag      .equ      0
0000                0036
0000                0037            inc_const inner,5
0000 05             0038            .byte     inner
0001                0039
0001                0040            inc_const outer,8
0001 0D             0041            .byte     outer
0002                0042
0002                0043            .if       inner > outer
0002                0044            s_bool    iflag,"true"
0002                0045            .else
0002                0046            s_bool    iflag,"false"
0002                0047            .endif
0002                0048
0002                0049            q_bool    iflag
0002 66616C7365     0001 #             .string   "false"
```

Note that if a symbol component is accessed and the variable's string component is not a symbol in the symbol table, the result will be 0 and the assembler will issue an error.

7.4.2.3 Using Qualified Macro Variables

When a variable and its component are enclosed in colons, the value of the component is formatted into a string, and the string is substituted into the line. For example, assume the macro variable xxx has the following components:

string: ASSEMBLER
value: 9999
length: 9 (length of the string ASSEMBLER)

The following statements would be translated as shown:

Source		Translation
mov	:xxx:,A	mov ASSEMBLER,A
mov	:xxx.s:,A	mov ASSEMBLER,A
mov	:xxx.v:,A	mov 9999,A
mov	:xxx.l:,A	mov 9,A

Note that the string component is always the default component. The assembler expands macro variables in model statements by using the string representation of the component. Thus, binary values such as the length and value components are formatted into strings and placed into the line. Colons are only necessary when no suffix is specified, or to delimit adjacent macro variable expansions.

In macro directive lines where a macro variable may appear in an arithmetic expression (.LOOP, .ASG, or .IF), the colons become significant. For example, the following two statements use the same macro variable xxx but produce different results.

```
.IF     xxx.v   == 9999
.IF     :xxx.v: == 9999
```

The first statement evaluates to true, since the binary value of the value component is used. In the second example, the string value of the value component (9999) is used. This is equivalent to the hex value 39393939_{16}, or four ASCII 9s.

7.4.2.4 Attribute Components

The assembler symbol table stores various types of information about symbols and variables (has it been defined, is it global, etc.). Each of these attributes is associated with a keyword. Two components allow you to use this attribute information:

- The **.A** component describes attributes of a macro variable.
- The **.SA** component provides information about a symbol in the assembler symbol table.

These components can be used in a macro expression in two ways (& is AND, | is OR):

`var.a & keyword`	Reads a value of the attribute
`var.a | keyword`	Sets a value in the attribute

These expressions return 1 for true and 0 for false, and must always appear in this exact format.

Note that setting an attribute in a symbol attribute does not change the real attributes of that symbol as they appear to the rest of the assembler. For example, you cannot change a symbol to a global symbol by setting its $def attribute. However, they can be used as flags between macros, if desired. Table 7-3 lists the valid keywords.

Table 7-3. Keywords

Macro Variable Attribute (.a Component) Keywords			
$pa	Set if the operand is register A	**$paddr**	Set if the operand is an address
$pb	Set if the operand is register B	**$pp**	Set if the operand is a peripheral register
$pr	Set if the operand is a register	**$psp**	Set if the operand is the stack pointer
$pst	Set if the operand is the status register	**$psub**	Set if the operand is a subscript
$pstr	Set if the operand is a string	**$pval**	Set if the operand is a value
$pw	Set if the operand is a work register	**$pcall**	Set if the variable was passed as an argument to the macro
$popl	Set if the parameter is an operand list (op, op, ...), note that when a list is passed to a macro, the value component contains the number of operands in the list		
Symbol Attribute (.sa Component) Keywords			
$def	Set if the symbol is global and defined	**$rel**	Set if the symbol is relocatable
$str	Set if the symbol has been assigned a string component	**$undef**	Set if the symbol is undefined

Example 7-5 (page 7-14) shows an example that uses variable attributes.

Using the attribute component in a model statement has no effect – it is only useful in macro directive lines. Note that whenever the attribute component of a variable is accessed in an expression, it must appear in an expression as in the examples above, with one of the two legal operators and an appropriate keyword. Any other use is illegal and will be flagged as an error.

Example 7-5. Using Variable Attributes

```
0000                    0001 keys  .MACRO   pitch
0000                    0002       .VAR     msg
0000                    0003       .ASG     '"':pitch.s: , pitch.s
0000                    0004 ;
0000                    0005       .IF      pitch.a & $PCALL
0000                    0006       .ASG     'is a parameter"' , msg.s
0000                    0007       .ELSE
0000                    0008       .ASG     'is not a parameter"' , msg.s
0000                    0009       .ENDIF
0000                    0010       .string  :pitch.s: :msg.s:
0000                    0011 ;
0000                    0012       .IF      pitch.a & $PR
0000                    0013       .ASG     'is a register"' , msg.s
0000                    0014       .ELSE
0000                    0015       .ASG     'is not a register"' , msg.s
0000                    0016       .ENDIF
0000                    0017       .string  :pitch.s: :msg.s:
0000                    0018 ;
0000                    0019       .IF      pitch.sa & $REL
0000                    0020       .ASG     'is relocatable"' , msg.s
0000                    0021       .ELSE
0000                    0022       .ASG     'is absolute"' , msg.s
0000                    0023       .ENDIF
0000                    0024       .string  :pitch.s: :msg.s:
0000                    0025 ;
0000                    0026       .IF      pitch.sa & $DEF
0000                    0027       .ASG     'is defined in this module"' ,   msg.s
0000                    0028       .ELSE
0000                    0029       .ASG     'is not defined in this module"  ' ,msg.s
0000                    0030       .ENDIF
0000                    0031       .string  :pitch.s: :msg.s:
0000                    0032 ;
0000                    0033       .ENDM
0000                    0034 ;
0056                    0035 CALL  .equ     56h
0000                    0036       keys     @CALL
0000 4043414C4C 0001 #             .string  "@CALL is a parameter"
0014 4043414C4C 0002 #             .string  "@CALL is not a register"
002B 4043414C4C 0003 #             .string  "@CALL is absolute"
003C 4043414C4C 0004 #             .string  "@CALL is defined in this module"
```

7.5 Manipulating Strings

When a model line is expanded, after all macro variables have been replaced, adjacent strings are concatenated. Example 7-6 shows an example that manipulates strings.

Example 7-6. Manipulating Strings

```
str_3   .MACRO   parm1,parm2,parm3
        .VAR     quote
        .ASG     '"' , quote
        .string  :quote.s::parm1: :parm2: :parm3::quote.s:
        .ENDM

        str_3    Great Lakes:,(Huron, Superior,),(Michigan, Erie, Ontario)
           .string  "Great Lakes: Huron, Superior, Michigan, Erie, Ontario"
```

This example concatenates three parameters:

- Parameter 1 is `Great Lakes:`.
- Parameter 2 is an operand list, `(Huron, Superior)`.
- Parameter 3 is an operand list, `(Michigan, Erie, Ontario)`.

7.6 Conditional Blocks

The .IF, .ELSE, and .ENDIF directives are used to construct conditional blocks within macro definitions. Conditional blocks may be nested up to ten levels deep. Blocks at all nesting levels must always be terminated with an .ENDIF. The general format of a conditional block is:

```
.IF   expression

code to assemble if expression is true (≠ 0)

.ELSE

code to assemble if expression is false (= 0)

.ENDIF
```

The relational operators that can be used in an .IF expression include:

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

These operations are all unsigned. They have the lowest precedence of any operations, but have equal precedence with each other and thus are evaluated from left to right. They evaluate to 1 if true and 0 if false, and the result of a relational expression can be used with any of the operators.

If the expression in the .IF statement evaluates to a nonzero value, then the code that follows it (up to an .ELSE or .ENDIF) will be assembled. If the expression evaluates to 0, then the assembler will not assemble the code that follows the .IF statement; if an .ELSE directive is present, the assembler will assemble the code that follows it (up to the .ENDIF).

All directives (.IF, .ELSE, and .ENDIF) in a single conditional block *must appear in the same source module.* For example, the .ENDIF cannot appear in an included file. A conditional block not terminated by the end of a source file (or upon encountering an .ENDM directive) will produce an error.

In a block of code that is not being assembled, include files and macro definitions are not scanned. Conditional assembly directives that appear in a macro definition are evaluated each time the macro is expanded, not as it is defined.

Example 7-5 (page 7-14) contains an example of conditional blocks.

7.7 Repeatable Blocks

Repeatable blocks allow a section of code (or a section of a macro definition) to be repeatedly expanded. This is particularly useful for table generation. The format of a repeatable block is:

```
.LOOP     expression

model statements
or
macro directives

.ENDLOOP
```

The assembler evaluates the expression once when it enters the loop, and then it repeats the block *expression* number of times. The expression may be any legal expression or macro expression.

The same restrictions apply to the declaration of a repeatable block as apply to conditional blocks. Blocks (regardless of block type) may be nested up to 10 deep. Improper nesting of blocks is checked by the assembler, and all errors are flagged. The following example shows improper nesting:

```
.LOOP     expression 1
  .
  .
.IF       expression 2
.ENDLOOP
  .
  .
.ENDIF
```

Note that the two blocks overlap rather than nest properly. This is an error, and the macro definition will be ignored.

Example 7-4 (page 7-11) shows an example of a repeatable block.

8. Archiver Description

The TMS370 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 file, you can use the archiver to add more files to it, delete or replace existing members, or extract members.

You can use the archiver to build libraries out of any type of files. Both the assembler and the linker can use archive libraries; 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 the 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. Section 7 discusses macros and macro libraries in detail.

This section contains the following topics.

Figure 8-1 (page 8-2) shows the archiver's role in the assembly language development process.

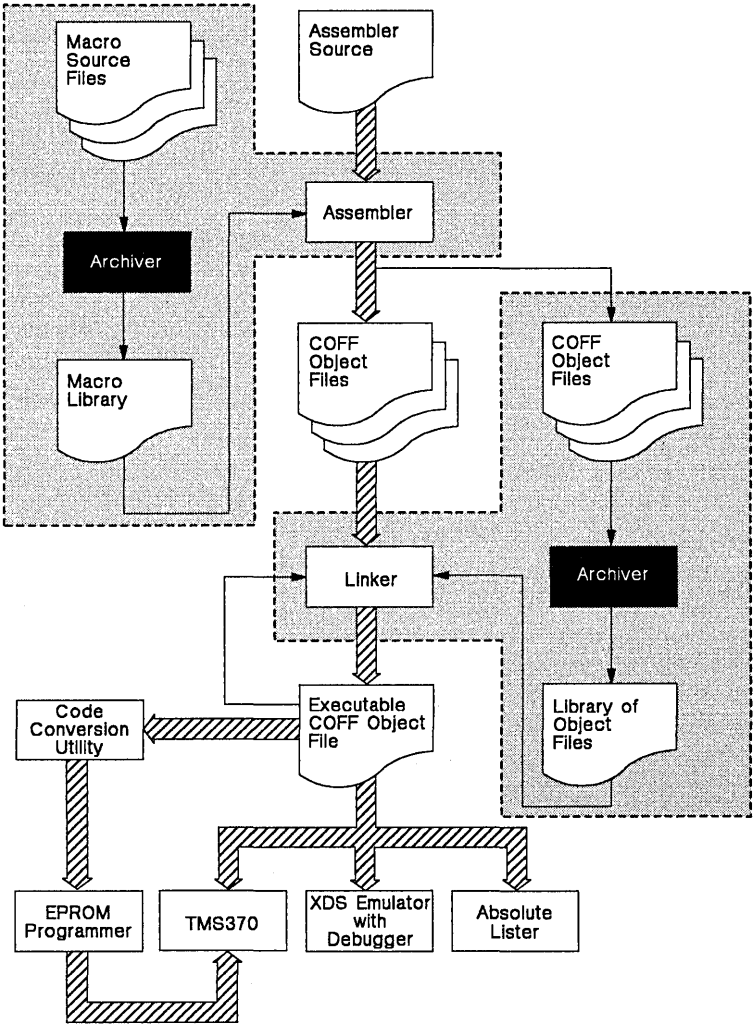


Figure 8-1. Archiver Development Flow

8.1 Invoking the Archiver

To invoke the archiver, enter:

```
ar370 [-]<command> <libname> <file 1> ... <file n>
```

Libname is the name of the archive library. If you don't specify an extension, the archiver will use the default extension **.lib**. The *members* are the names of the individual member files contained in the library.

The *command* tells the archiver how to manipulate the members in the library. You must use only **one** command per invocation. A command can be preceded by an optional hyphen. Valid archiver commands include:

a Add the specified files to the library. Note that this command **does not replace** existing members that have the same name as an added file; it simply *appends* new members to the end of the archive. If you want to *replace* existing members, use the r command.

d Delete the specified members from the library.

r Replace the specified members in the library. If you don't specify any member names, the archiver will replace the members with files of the same name in the current directory. If the specified file is not found in the archiver, it is added instead of replaced.

t Print a table of contents of the library. If no names are given, all files in the archive are listed. If names are given, only those files are listed.

x Extract the specified files. If you don't specify any member names, the archiver will extract all the files in the library. When the archiver extracts a file, it simply puts a copy of it in the current directory; it doesn't alter the library.

In addition to the commands listed above, you can specify one or both of the following options:

s Print a list of the symbols that are defined in the library.

v (verbose) The archiver will provide a file-by-file description of the creation of a new library from an old library and its constituent members.

8.2 Archiver Examples

Here are some examples of using the archiver.

- **Example 1:**

  If you wanted to create an archive file, you could specify:

  ```
  ar370 -a function sine.obj cos.obj flt.obj
  ```

  This would create a library called function.lib. It would contain the files sine.obj, cos.obj, and flt.obj.

- **Example 2:**

  Since no extension was specified for the libname in first example, the archiver used the default extension .lib. You can, however specify a different extension:

  ```
  ar370 -a function.fn sine.obj cos.obj flt.obj
  ```

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

- **Example 3:**

  If you wanted to add some new members to a library, you would specify:

  ```
  ar370 -a function tan.obj arctan.obj area.obj
  ```

  Since this example doesn't specify an extension for the libname, the archiver adds the files to the file called function.lib. If function.lib didn't exist, the archiver would create it. In this example, the library function.lib contains the files tan.obj, arctan.obj, and area.obj, as well as sine.obj, cos.obj, and flt.obj (which were put in the library in the first example).

- **Example 4:**

  If you want to modify a member of 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.

  ```
  ar370 -x macros push.asm
  ```

  The archiver will make a copy of push.asm and place 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 copy that was changed, enter:

  ```
  ar370 -r macros push.asm
  ```

9. Linker Description

The TMS370 linker creates executable modules by combining COFF object files. The concept of COFF *sections* is basic to linker operation; Section 3 discusses COFF sections in detail. The linker accepts several types of files as input:

- Relocatable COFF object files produced by the TMS370 assembler,
- Command files,
- Archive object libraries, **and**
- Output modules created by a previous linker run (these are referred to as *partially linked* files).

Figure 9-1 illustrates the linker's role in the assembly language development process. As the linker combines object files, it performs the following tasks:

- It allocates sections into the target system's configured memory.
- It relocates symbols and sections to assign them to final addresses.
- It resolves undefined external references between input files.

The linker supports a C-like command language that controls memory configuration, 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, **and**
- Define or redefine global symbols at link time.

Topics in this section include:

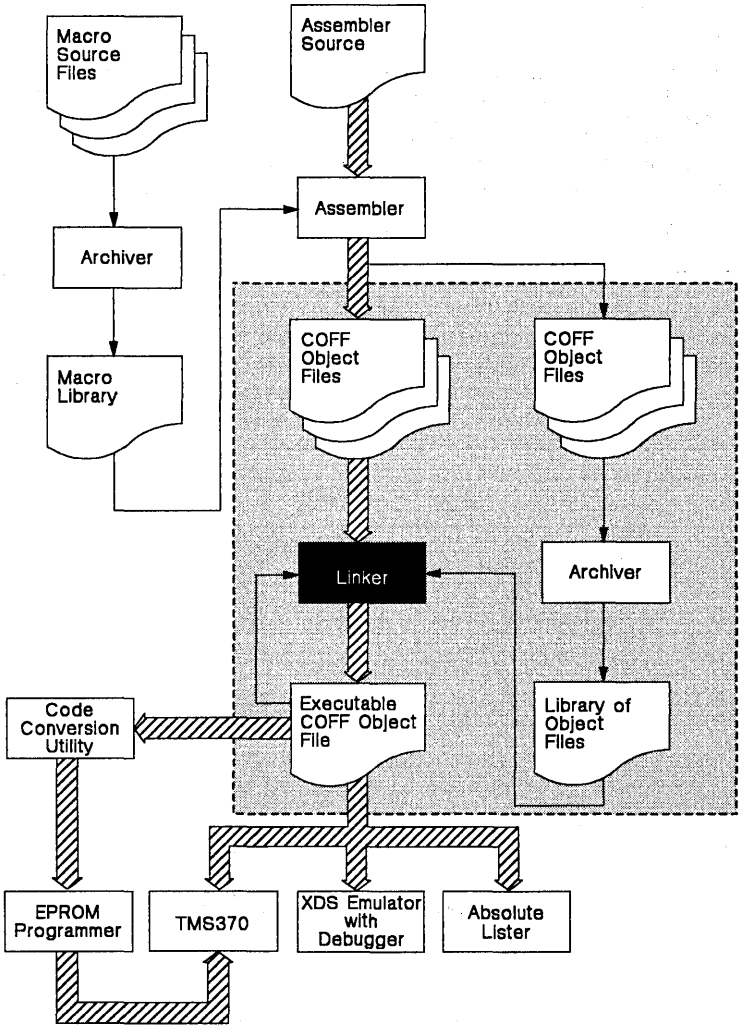


Figure 9-1. Linker Development Flow

9.1 Invoking the Linker

The general syntax for invoking the linker is:

```
lnk370  [-<options>] <filenames>
```

where *options* (discussed in Section 9.2) can appear anywhere on the command line or in a linker command file. The *filenames* can be object files, linker command files, or object 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 module name 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`.

  ```
  lnk370 -o link.out file1.obj file2.obj
  ```

- Enter the **lnk370** command with no filenames or 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 a default filename consisting of the first object file name with the extension **.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: `lnk370 linker.cmd`.

  When you use a command file, you can also specify other options and files that are not listed in the command file. For example, you could enter: `lnk370 -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`.

9.2 Linker Options

Linker options control linking operations. Options can be placed on the command line or in a command file. All options must be preceded by a hyphen (-). The order in which options are specified is unimportant, except for the -l and -L options. Options are separated from arguments (if they have them) by a space (except, again, for -l and -L). Table 9-1 summarizes the linker options.

Table 9-1. Linker Options Summary

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 was specified.
-e *global symbol*	Define an entry point (named by *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 2-byte constant.
-H	Make all global symbols static.
-l*filename*†	Name an archive library file as linker input. *Filename* is an archive library name.
-L*dir*	Alter the library-search algorithm to look in *dir* before looking in the default location. This option must appear before the -l option.
-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*.
-r	Retain relocation entries in the output module.
-s	Strip symbol table information and line number entries from the output modules.
-S	Request a "silent" run; that is, -S suppresses all error messages that do not halt linker execution.
-u *symbol*	Place an unresolved external symbol, *symbol*, into the output module's symbol table.

† *Filename* is a filename that follows operating system conventions.

9.2.1 Relocation Capability (-a and -r Options)

One of the tasks the linker performs is *relocation*. Relocation is the process of adjusting all the references to a symbol when the symbol's address changes.

For example, a program may contain an instruction that jumps to address x. Assume that when the program was assembled, x had a value of 5; thus, the assembler created an instruction to "jump to 5." The assembler also put a relocation entry in the object file that identified the fact that the jump is actually to symbol x. When the program is linked, assume that x's end location is now 100. The linker uses the relocation entry to change the "jump to 5" to "jump to 100."

The linker supports two options (**-a** and **-r**) that allow you to choose whether you will produce an absolute or a relocatable output module.

- Producing a Relocatable Output Module

  The **-r** option tells the linker to retain relocation entries and produce a relocatable 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. When -r is used, unresolved references do not prevent creation of the output module.

  This example links `file1.obj` and `file2.obj` and creates a relocatable output module called `a.out`:

  ```
  lnk370 -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*. For more information about partial linking, see Section 9.11, page 9-35.)

- Producing an Absolute Output Module

  The **-a** option produces an absolute executable output module. Relocation entries are stripped from absolute files; thus, absolute files can only be relocated (at load time) or relinked (by the linker) under special circumstances.

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

  ```
  lnk370 -a file1.obj file2.obj
  ```

- 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 only be successful if each input file contains no information that needs to 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).

> **Note:**
>
> If neither the -a nor the -r option are specified, the linker acts as if -a is specified.

9.2.2 Defining an Entry Point (-e <global symbol> Option)

An output module contains a field that identifies the beginning execution address in target memory. This address (called the **entry point**) is used when the file is loaded to initialize the program counter to point to the beginning of a 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. Note that the first three values require a symbol to be placed in the symbol table by the .global assembler directive. These symbols are case sensitive.

Possible entry point values include:

1) 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 to be linked.

2) The value of symbol `_main` (if present).

3) Zero (default value).

This example links `file1.obj` and `file2.obj` and sets the entry point to the value of the symbol `begin`. This symbol must be defined as external in `file1` or `file2`.

```
lnk370 -e begin file1.obj file2.obj
```

9.2.3 Set Default Fill Value (-f <cc> Option)

The -f option fills the holes formed within output sections or initializes .bss sections when they are combined with non-.bss sections. This allows you to initialize memory areas during link time without reassembling a source file.

The argument *cc* is a 2-byte constant (up to four hexadecimal digits). When -f is not used, the default fill value is 0.

This example fills holes with the hexadecimal value ABCD:

```
lnk370 -f 0ABCDh file1.obj file2.obj
```

9.2.4 Make All Global Symbols Static (-H Option)

The -H option makes all global symbols static. This "hides" symbols, because static symbols are not visible to externally linked modules. This allows external symbols with the same name (in different files) to be treated as unique.

The -H option effectively nullifies all .global assembler directives. All symbols become local to the module in which they were defined, so no external references are possible.

For example, assume `file1.obj` and `file2.obj` both define global symbols called `ext`. By using the -H option, these files can be linked without conflict. The symbol `ext` defined in `file1.obj` is treated separately from the symbol `ext` defined in `file2.obj`.

```
lnk370 -H file1.obj file2.obj
```

9.2.5 Specify a Directory and an Archive Library (-L <dir>, -l<filename> Options)

The -L option alters the library search algorithm. When the linker is searching for library members, it will search the directory *dir* before it searches the current directory. The -l (lowercase "L") option names a specific library that should be taken from directory *dir*. The *filename* is an archive library filename. It is not always necessary to use the -l option when specifying a library as linker input; -l is only useful when you specify a library that is in a directory named with -L. Note that no space separates the -L or -l options and their arguments. If the -L option is used with the -l option, -L must be specified first.

As an example, assume that an archive library called `rts.lib` resides in a directory called `\libdir`. Assume that another library called `lib2.lib` resides in a directory called `\libdir2`. You can use both libraries during a link by specifying the following:

```
lnk370  file1.obj file2.obj -L\libdir -L\libdir2 -lrts.lib -llib2.lib
```

(This example is for a PC/MS-DOS system.)

9.2.6 Create a Map File (-m <filename> Option)

The -m option writes a link map or listing to *filename*. This map describes:

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

The map file contains the name of the output module, the entry point, and 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 which comprise 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, 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`:

```
lnk370 file1.obj file2.obj -m map.out
```

Section 9.12 (page 9-36) contains an example of a map file.

9.2.7 Naming an Output Module (-o <filename> Option)

The linker always creates an executable output module. If you do not specify a filename for the output module, the linker assigns it the default name of `a.out`.

If you want to write the output module to another file, use the -o option. The argument *filename* is the new output module name.

This example links `file1.obj` and `file2.obj` and writes the resulting output module to the file `run.out`:

```
lnk370 -o run.out file1.obj file2.obj
```

9.2.8 Stripping 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 places the output module, stripped of line numbers and symbol table information, in the file `nolink.out`:

```
lnk370 -o nolink.out -s file1.obj file2.obj
```

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

9.2.9 Specifying a Silent Run (-S Option)

The -S option suppresses all messages caused by diagnostic warnings and by errors that are not fatal; that is, they do not halt linker execution.

9.2.10 Introduce an Unresolved Symbol (-u <symbol> Option)

The -u option introduces an unresolved *symbol* into the linker's symbol table. This is useful for forcing the linker to search a library in order to resolve a symbol.

For example, suppose a symbol `symtab` is defined in a module in an archive library named `rts.lib`. None of the other object files being linked refer to `symtab`. However, suppose this file will be relinked, and you would like to include the module that defines `symtab`. By using the -u option, as shown below, the linker is forced to search `rts.lib` for the module that defines `symtab` and to include the module.

```
lnk370 -u symtab file1.obj file2.obj rts.lib
```

If you did not use -u, this module would not be included since there is no explicit reference to it in `file1.obj` or `file2.obj`.

9.3 Linker Command Files

Linker command files allow you to specify the linking parameters 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. These directives must be used in a command file; there is no way to specify them from the command line. Command files can contain one or more of the following:

- Input filenames, which specify object files, archive files, or other command files.
- Options, which can be used in the command file in the same manner that they are used on the command line.
- Linker directives, which include the MEMORY and SECTIONS directives. The MEMORY directive allows you to specify the target memory configuration. The SECTIONS directive controls how sections are built and allocated.
- Assignment statements, which define and assign values to global symbols.

9.3.1 Command File Format

Command files are ASCII files that contain filenames, options, and linker directives. You can supply a command file to the linker by specifying it on the command line in the same manner that you would specify an object file. The linker processes input files in the order that they are encountered. If the linker recognizes a file as an object file, it links it. Otherwise, it assumes a file is a command file, and begins reading and processing commands from it.

Here is 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     */
```

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 using this command file, enter:

```
lnk370 link.cmd
```

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

```
lnk370 -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 also 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:

```
lnk370 names.lst dir.cmd
```

One command file can also call another command file; this type of nesting is limited to 16 levels.

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. Here is a sample command file that contains linker directives:

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

MEMORY                      /* MEMORY Directive          */
{
  RAM:  o = 100h      l = 0100h
  ROM:  o = 01000h    l = 0100h
}
SECTIONS                    /* SECTIONS Directive        */
{
  .text: {} > ROM
  .data: {} > ROM
  .bss:  {} > RAM
}
```

(Linker directive formats are discussed in later sections.)

9.3.2 Names Reserved for the Linker

The linker directives use the following names, so these names are reserved. Do not use them as symbol or section names in a command file.

align	len	origin
ALIGN	length	ORIGIN
block	LENGTH	page
BLOCK	MEMORY	PAGE
DSECT	NOLOAD	range
group	o	SECTIONS
GROUP	org	spare
l		

9.4 Archive Libraries

An archive library is a partitioned file that contains complete object files as members. Usually, a group of related modules are grouped together into an archive library. When you specify an archive library as linker input, the linker will include any members of the library that define existing undefined symbol references. You can use the TMS370 archiver to build and maintain archive libraries; Section 8 discusses the archiver.

Libraries can be useful for reducing link time and reducing the size of the executable module. If a normal object file that contains a function is specified at link time, it will be linked whether it is used or not; however, if that same function is placed in a library, it will only be included if it is referenced.

The order in which archive 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 will be searched each time it is included. An archive file has a symbol table that contains all external symbols that are resolved in the library; the linker searches through the table until it determines it cannot use the library to resolve any more references.

Assume the following:

- Input files `f1.obj` and `f2.obj` both reference an external function `clrscr`.
- Input file `f1.obj` references the symbol `origin`.
- Input file `f2.obj` references the symbol `fillclr`.
- Library `liba.lib`, member 0, contains a definition of `origin`.
- Library `libc.lib`, member 3, contains a definition of `fillclr`.
- Both libraries have a member 1 that defines `clrscr`.

If you enter: `lnk370 f1.obj liba.lib f2.obj libc.lib`

then:

- 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 `liba.lib` satisfies the reference to `origin`.
- Member 3 of `libc.lib` satisfies the reference to `fillclr`.

If, however, you enter: `lnk370 f1.obj f2.obj libc.lib liba.lib`

then the references to `clrscr` are satisfied by member 1 of `libc.lib`.

If none of the files being linked reference symbols defined in a library, you can use the -u linker 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:

```
lnk370 -u rout1 libc.lib
```

If any members of `libc.lib` define `rout1`, then the linker will include those members. Note that there is no method for identifying specific members in an archive library. As a result, the sections of an archive member are allocated into the output module using the default allocation rules; there is no method for overriding this allocation with the SECTIONS directive.

9.5 The MEMORY Directive

One of the linker's tasks is to decide where in physical memory each section of the output module will be loaded. This process is called **allocation**. To perform allocation, the linker must comprehend the target system's memory configuration.

The TMS370 architecture supports multiple address spaces; the system memory configuration can change at run time in response to I/O signals in the device. As a result, different banks of physical memory may be mapped into a single address range at different times. To the linker, each possible memory configuration represents a separate address space. Each address space is called a *page* and must be configured separately.

The MEMORY directive allows you to specify a model of target memory, so you can specify the particular sections that should be loaded into various banks of physical memory. The linker maintains the model as it allocates output sections, and uses it to determine which locations in the target system can be used for the linked program.

9.5.1 Default Memory Model

The linker's default memory model is based on the TMS370 architecture. This model assumes that the following memory is available:

- 254 bytes of RAM, beginning at location 02h (the register file)
- 256 bytes of EEPROM, beginning at location 1F00h
- 4K bytes of ROM, beginning at location 7000h

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

9.5.2 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:

- A name
- A starting address
- A length
- An optional set of attributes

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 will 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
{
  RFILE:   origin =    02h  length =  0FEh
  EEPROM:  origin =  1F00h  length =  100h
  ROM:     origin =  7000h  length = 1000h
}
```

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

The general syntax for the MEMORY directive is:

```
MEMORY
{
  name1 [(attr)] : origin = constant , length = constant
  name2 [(attr)] : origin = constant , length = constant
       .
       .
       .
}
```

(Underscored items must be entered as shown.)

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 that are associated with the named range. Attributes are optional; when used, they must be enclosed in parenthesis. 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. It may be abbreviated as *org* or *o*. The value, specified in bits, is a long integer constant, and may be decimal, octal, or hexadecimal.

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

Here is an example that specifies a memory range with the R and W attributes:

```
MEMORY
{
      RFILE (RW) : o = 02h, l = 0FEh
}
```

You will usually 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.

9.5.3 Checking the Results of the MEMORY Directive

The linker builds a table of the memory model as specified by the MEMORY directive. It puts this table in the map file, providing you with an easy method to check the results of the MEMORY directive. To obtain a map file, invoke the linker with the -m option:

```
lnk370 -m <map file name>
```

Section 9.12 (page 9-36) contains an example of a map file.

9.6 The SECTIONS Directive

The SECTIONS directive tells the linker how to combine sections in 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 will be combined into output sections,
- Defines output sections in the executable program,
- Specifies where output sections will be placed in memory (in relation to each other and to the entire memory space), **and**
- Permits renaming of output sections.

9.6.1 Default Sections Configuration

If you do not specify a SECTIONS directive, the linker uses a default algorithm for combining and allocating the sections. Section 9.8 describes this algorithm in detail.

9.6.2 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.

Here is an example of the SECTIONS directive:

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

SECTIONS
{
   .text 07000h : { }

   .data : { file1.obj(.data) }

   init  :
           {
             file1.obj(init)
             file2.obj(.data)
           }

   .bss  ALIGN(16) : { }
}
```

This example defines four output sections, .text, .data, init, and .bss:

- The **.text** output section is composed of the .text sections from `file1.obj` and `file2.obj`. Notice that the braces ({ }) are empty in this section specification; that tells the linker to include all input sections that have the same name as the output section.

  Binding was specified for this output section; so, the .text output section will begin at address 07000h in the target memory.

- The **.data** output section contains the .data section from `file1.obj`.
- The **init** section is composed of the init (named) section in `file1.obj` and the .data section in `file2.obj`.
- The **.bss** output section is composed of the .bss sections from `file1.obj` and `file2.obj`. This output section will be aligned on the next available 16-byte boundary.

The general syntax of the SECTIONS directive is:

```
SECTIONS
{
        section specification 1
        section specification 2
        section specification n
}
```

Each section specification defines an output section. (An output section is simply a section in the final executable output file.) The syntax for a section specification is:

```
name     [ binding or align(n) ] :
{
     input sections
     assignments
} [ = fill value ] [ > named memory ]
```

(Underscored portions must be entered as shown.)

name Specifies the name of the section in the output file. A name may have up to eight characters.

binding Is optional, and assigns the section to a specific physical address in the target memory. Section 9.6.4 discusses assigning an address to an output section.

alignment Is optional, and specifies that the section should be aligned on an address boundary (the actual address is determined by the linker). Section 9.6.4 discusses aligning an output section.

input sections Is a list of input sections that are combined to form the output section. The list is enclosed in braces. Section 9.6.3 discusses specifying input sections in detail.

assignment Is optional, and defines the value of symbols at link time or creates uninitialized spaces (called holes) between input sections within the output section. See Section 9.10 for more information about holes.

fill value Is optional, and specifies a value for filling holes in the section. See Section 9.10 for more information about fill values for holes.

>named memory Is optional, and specifies that an output section should be allocated into a memory range that was named by the MEMORY directive. Section 9.6.4 discusses named memory.

9.6.3 Specifying Input Sections

The input sections specifications in the SECTIONS directive specify which sections from input files are combined to form an output section. The linker combines input sections by concatenating them in the order in which they are specified. The size of an output section is the sum of the sizes of the input sections that make up the output section.

Normally, an output section specification lists no input sections:

```
.text : { }
```

In this case, the linker takes all the .text sections from the input files and combines them into the .text output section. It concatenates them in the order in which the input files were specified to the linker. You can, however, explicitly specify the input sections that will form an output section. Each input section is identified by its filename and section name:

```
SECTIONS
{
  .text :
  {
    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 input sections to have the same name as each other, or of the output section they become part of. If a file is listed with no sections, **all** of its sections are included in the output section. If there are any additional input sections that have the same name as the output section, but are not explicitly specified by the SECTIONS directive, they will automatically be 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, then the linker would concatenate these extra sections after file4.obj(sec2).

9.6.4 Specifying the Address of Output Sections (Allocation)

After the linker has determined the composition of each output section, it must determine where in physical memory the section will be loaded. Each section has an address field in its section header that tells the loader where the linker decided the section should go. The process of calculating the address of the output sections is called *allocation*.

If you do not specify an explicit starting address for an output section, the linker will use a default algorithm to allocate the section. Generally, the linker puts sections where ever they will fit into configured memory.

You can override this default allocation by telling the linker where the section should be loaded. You can use three methods to control section allocation:

- **Binding**

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

  ```
  .text 01000h : { ... }
  ```

  This example specifies that the .text section must begin at location 1000h. The binding address must be a 16-bit decimal, octal, or hexadecimal 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 will issue an error message.

  Note that you cannot use the binding method if you also use alignment or named memory. If you try to do this, the linker will issue an error message.

- **Alignment**

  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,

  ```
  SECTIONS
  {
      .data  ALIGN(32) : { ... }
  }
  ```

  In this example, the .data output section is not bound to a specific address; it is linked at the next available address in configured memory that is a multiple of 32 bytes.

- **Named Memory**

  A section can be linked into a memory range that has been named by the MEMORY directive. This example names ranges and links sections into them.

  ```
  MEMORY
  {
      ROM (RIX) :  origin = 7000h, length = 1000h
      RAM (RWIX):  origin = 8000h, length = 1000h
  }

  SECTIONS
  {
      .text :             { ... } > ROM
      .data ALIGN(64) :   { ... } > RAM
      .bss  :             { ... } > RAM
  }
  ```

  In this example, the linker places .text into the area called ROM, between locations 7000h and 7FFFh. The .data and .bss sections are placed into RAM. It is possible to align a section within named memory; the .data section is aligned on a 64-byte boundary.

Similarly, you can specify that a section be linked into an area of memory that has particular attributes. To do this, specify a set of attributes (enclosed in parenthesis) instead of a memory name. Assuming you used the same MEMORY directive declaration, you could specify:

```
SECTIONS
{
  .text: {...} > (X)     /* .text --> executable memory     */
  .data: {...} > (RI)    /* .data --> read and init memory  */
  .bss : {...} > (RW)    /* .bss  --> read and write memory */
}
```

In this example, the .text section can be linked into either the ROM or RAM area, since both were declared with the X attribute. The .data section can also go into either ROM or RAM, since both have the R and I attributes. The .bss section, however, must go into the RAM area, because only RAM was declared with the W attribute.

You cannot control where in the memory range the section will be 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 7000h. If a section must start on a specific address, use binding instead.

9.6.5 Grouping Output Sections Together

The SECTIONS directive has a GROUP option that forces specified output sections to be allocated contiguously. This prevents an output section from being fragmented when it is loaded.

For example, assume there is a section named term_rec that contains a termination record for a table in the .data section. You can force the linker to allocate term_rec next to .data as follows:

```
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 specify the allocation process for a GROUP in the same way as an output section. In the preceding example, the GROUP was bound to address 1000h. This means that .data will be allocated at 1000h, and term_rec will immediately follow it in memory. You can also use alignment and named memory with the GROUP option.

> **Note:**
>
> When you use the GROUP option, binding, alignment, or assignment to named memory can be specified for the group only. You cannot specify addresses for sections within a group.

9.6.6 Checking the Results of the SECTIONS Directive

The linker builds a table of all the output sections, their final allocated addresses, their sizes, and the address and size of each input section within the output section. It puts this table in the map file, providing you with an easy method to check the results of the SECTIONS directive. To obtain a map file, invoke the linker with the -m option:

```
lnk370 -m <map file name>
```

Section 9.12 (page 9-36) contains an example of a map file.

9.7 Overlay Pages

In some TMS370 applications, the memory architecture of the system can change at run time in response to I/O signals on the device. As a result, different banks of memory may be mapped into a single address range at different times. This means that multiple areas of physical memory overlay each other at one address space. You may want the linker to load various output sections into each of these areas.

The linker supports this feature by providing **overlay pages**, allowing you to define a memory model that has multiple address spaces. Each address range is treated as a separate page, and must be configured separately with the MEMORY directive. You can then use the SECTIONS directive to specify which sections will be mapped into various pages.

9.7.1 Using the MEMORY Directive to Define Overlay Pages

Each separately configured address space is called a *page*. To the linker, each page represents a completely separate memory. This allows you to link two or more sections at the same (or overlapping) addresses *if they are on different pages*.

Pages are numbered sequentially, beginning with 0. Page 0 represents the "normal" address space of the TMS370. The default memory model resides entirely on page 0. If a memory range is specified without a page number, the linker assumes it is on page 0. This allows you to ignore the page feature for normal cases; everything can be linked in page 0 with no overlays.

For example, suppose you are running the TMS370 in expanded microcomputer mode and using the address map hardware to select from four banks of external expansion memory. Suppose that there are four different output sections called `sect0`, `sect1`, `sect2`, and `sect3`, that must be linked into the four banks of memory. Expansion memory occupies 32K bytes of the address space, beginning at location 8000h. This is how you would use the MEMORY directive to obtain this configuration:

```
/*****************************************************/
/*** Example of MEMORY directive with overlay pages ***/
/*****************************************************/
MEMORY
{
   PAGE 0:  RFILE   : origin =   02h    length =  0FEh
            EEPROM  : origin = 1F00h    length =  100h
            ROM     : origin = 7000h    length = 1000h
            EXP     : origin = 8000h    length = 8000h
   PAGE 1:  EXP     : origin = 8000h    length = 8000h
   PAGE 2:  EXP     : origin = 8000h    length = 8000h
   PAGE 3:  EXP     : origin = 8000h    length = 8000h
}
```

Figure 9-2 (page 9-22) illustrates this configuration; it shows each available block of physical memory in the system and the section that must be loaded into it.

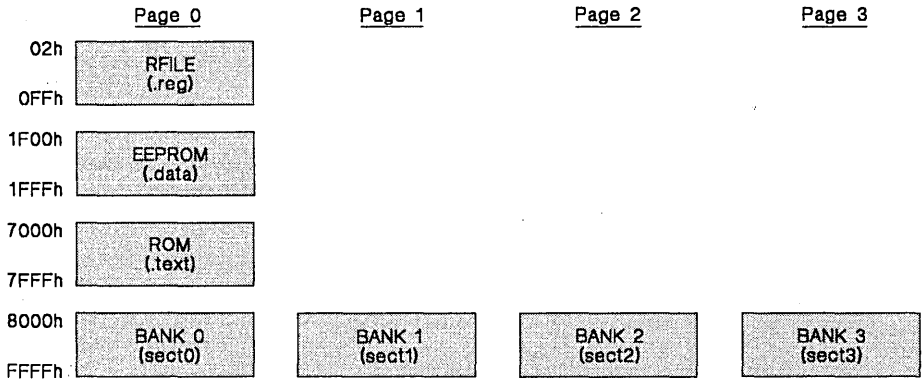


Figure 9-2. Overlay Page Example

This example defines four separate address spaces. Page 0 is the "normal" address space of the TMS370. It contains the memory ranges RFILE, EEPROM, ROM, and the first bank of expansion memory (EXP). The other three address spaces contain only the additional banks of overlay memory, all labeled EXP. Note that all four EXP ranges cover the same address range. This is possible because each range is on a different page, and therefore represents a different memory space.

9.7.2 Using Overlay Pages with the SECTIONS Directive

The SECTIONS directive allows you to tell the linker which page an output section should be linked into. Each output section of the program is assigned a page as well as an address. You can assign an output section to an overlay page by following the section specification with the PAGE option and a page number. Continuing the example from the previous discussion, the SECTIONS directive would be written as follows:

```
SECTIONS
{
   .reg : {} > RFILE              /* Link .reg at 0 in page 0          */
   .data: {} > EEPROM             /* Link .data at 1F00h in page 0     */
   .text: {} > ROM                /* Link .text at 7000h in page 0     */
   sect0: {} > EXP PAGE 0         /* Link sect0 into bank 0 (page 0)   */
   sect1: {} > EXP PAGE 1         /* Link sect1 into bank 1            */
   sect2: {} >     PAGE 2         /* Link sect2 into bank 2            */
   sect3: {} >     PAGE 3         /* Link sect3 into bank 3            */
}
```

If you don't specify a page number for an output section, the linker assumes page 0. In this example, .reg, .text, and .data are all linked into the named memory areas on page 0. (The PAGE 0 could have been omitted from the sect0 definition, and the same effect would be achieved.)

The PAGE specification for sect0, sect1, and sect2 tell the linker to link these output sections into the corresponding overlay pages. As a result, they all are linked to address 8000h, but *in different memory spaces*. When the program is loaded, the loader can configure hardware in such a way that each of these sections is loaded into the appropriate bank of memory.

Within a page, you can bind output sections to addresses or memory areas in the usual way. In the preceding example, notice how `sect1` is bound to the memory range called `EXP`. This allows you to define the allocation of sections within a page, just as you can in a single memory space. For example:

```
sect1 8F00h: {} PAGE 1
```

links `sect1` at address 8F00h in page 1. If you do not specify any binding or named memory range for the section, the linker allocates the section where ever it can into the page (just as it normally does with a single memory space). The definitions of `sect2` and `sect3` in the example illustrate this. Since `EXP` is the only memory on pages 2 and 3, it is not necessary (but acceptable) to specify `>EXP` for these sections.

9.7.3 Syntax of Page Definitions

As illustrated in the preceding examples, overlay pages are specified in the MEMORY directive by using the following syntax:

```
MEMORY
{
   PAGE  0  :  memory range
               memory range
    .
    .
   PAGE  n  :  memory range
               memory range
}
```

Each page is introduced by the keyword PAGE and a page number, followed by a colon and a list of memory ranges that comprise the page. Underscored portions must be entered as shown. Memory ranges are specified in the normal way. You can define up to 255 overlay pages.

Since each page represents a completely independent address space, memory ranges on different pages can have the same name. Configured memory on any page can overlap configured memory on any other page. *Within a single page, however, all memory ranges must have unique names and must not overlap.*

Any men ory ranges listed outside the scope of a PAGE specification default to page 0. Consider the following example:

```
MEMORY
{
           RFILE : org =   02h   len =  0FEh
           EEPROM: org = 1F00h   len =  100h
           ROM   : org = 7000h   len = 9000h
   PAGE 1: XRAM  : org = 2000h   len = 2000h
           XROM  : org = 8000h   len = 8000h
}
```

The memory ranges `RFILE`, `EEPROM`, and `ROM` are all on page 0 (since no page is specified). `XRAM` and `XROM` are on page 1. Note that `XROM` on page 1 overlays `ROM` on page 0.

In the output link map (obtained with the -m linker option), the listing of the memory model is keyed by pages. This provides you with an easy method of verifying that you specified the memory model correctly. Also, the listing of output sections has a PAGE column that identifies the memory space into which each section will be loaded.

9.8 Default Allocation Algorithm and Special Section Types

Using the MEMORY and SECTIONS directives provides you with a great deal of flexibility in specifying how sections will be built and combined. However, anything that you choose *not* to specify must still be handled by the linker. The linker has default algorithms that it uses to build and allocate sections, within the specifications you supply. Section 9.8.1 and Section 9.8.2 describe default allocation algorithms.

The linker also has the ability to create sections that are handled differently than the normal allocation algorithm specifies; Section 9.8.3 describes these sections.

9.8.1 Default Allocation Algorithm

If you do not use any MEMORY or SECTIONS directives, the linker acts as though the following definitions were specified:

```
MEMORY
{
   RFILE  : origin == 0002h   length =  0FEh
   EEPROM : origin == 1F00h   length =  100h
   ROM    : origin == 7000h   length = 1000h
}
SECTIONS
{
   .reg   : { }  > RFILE
   .bss   : { }  > RFILE
   .text  : { }  > EEPROM
   .data  : { }  > ROM
}
```

All .reg (relocatable register) input sections are concatenated to form one .reg output section, and all .bss input sections are concatenated to form one .bss output section. The .reg section is then linked into the register file, starting at location 02h (register R2), followed by the .bss section. All .data input sections are combined to form a .data output section, which is linked into EEPROM starting at location 1F00h. All .text input sections are concatenated to form a .text output section, which is linked into program memory starting at location 7000h.

Unless you specify otherwise with a MEMORY directive, the linker assumes the configuration specified above. That is, the only memory that the linker will use to build you program is:

- 254 bytes starting at location 02h
- 256 bytes starting at location 1F00h
- 4K bytes starting at location 7000h

If there are additional input sections in the input files (specifically, named sections), the linker will link 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 where ever there is room. Usually it is desirable to use explicit SECTIONS directives to tell the linker where to place named sections.

> **Note:**
>
> 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.

9.8.2 General Rules for Output Sections

An output section can be formed in one of two ways:

1) As the result of a SECTIONS directive definition.

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), its specification in the directive completely determines its contents. The contents of an output section in the SECTIONS directive is given by the information within the inner braces after the section name. (See Section 9.6 for examples of how to specify the contents of output sections.)

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 with 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 called `Vectors`. The linker will combine the two `Vectors` sections from the input files into a single output section named `Vectors`, allocate it into memory, and include 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 default configuration is used.

The linker uses an allocation algorithm that attempts to minimize memory fragmentation, which 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 listed 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) Any remaining sections are allocated in the order in which they were defined. Sections not defined in a SECTIONS directive are allocated in the order in which they were encountered. Each output section is placed in to the first available memory space, considering alignment where necessary.

> **Note:**
>
> If you do not use the PAGE option to explicitly specify a memory space for an output section, the linker will allocate the section into page 0. This will occur even if there is no room on page 0 but there is space on other pages. To use a page other than page 0, you **must** specify the page with the SECTIONS directive.

9.8.3 DSECT, COPY, and NOLOAD Sections

There are three special types that you can assign to output sections. These types affect the way that the program is treated when it is linked and loaded. These types are DSECT, COPY, and NOLOAD. A type may be assigned to a section by placing the type (enclosed in parenthesis) after the section definition. For example,

```
SECTIONS
{
        sec1 2000h (DSECT)  : {f1.obj}
        sec2 4000h (COPY)   : {f1.obj}
        sec3 6000h (NOLOAD) : {f1.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 identical 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: it is not written to the output module. It is allocated space, appears in the memory map listing, etc.

9.9 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.

9.9.1 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 to symbol
symbol	*=	expression;	Multiplies symbol by expression
symbol	/=	expression;	Divides symbol by expression

The symbol should be an externally defined symbol in your program. If it is not, the linker will define a new symbol and enters it into the symbol table. The expression must be a valid expression that follows the rules defined in Section 9.9.3. Assignment statements **must** be terminated with a semicolon.

The linker processes assignment statements after it allocates all the output sections. This means that if the expression assigned to a symbol contains a second symbol name, the address used for the second symbol in the expression reflects the symbol's address in the executable output module.

For example, suppose you have a program that can read data from one of two tables identified by the symbols `Table1` and `Table2`. Your program uses the symbol `cur_tab` as the address of the current table to read. Assume all these symbols are external (declared with the .global assembler directive). You must assign `cur_tab` to point to either `Table1` or `Table2`. You could do this in the assembly code, but this requires you to reassemble the program in order to change tables. Instead, you can use an assignment statement to assign cur_tab at link time with the following command file:

```
prog.obj          /* Input file                          */
cur_tab = Table1; /* Assign cur_tab to one of the tables */
```

9.9.2 Assigning the PC to a Symbol

A special symbol, denoted by a dot (.), represents the current value of the PC during allocation. The linker's "." symbol is analogous to the assembler's $ symbol The "." symbol can only be used in assignment statements within a SECTIONS directive, since "." is only meaningful during allocation and the allocation process is controlled by the SECTIONS directive.

For example, suppose your program needs to know the address of the beginning of the .data section. You can create an external undefined variable `Dstart` in your program by using the .global assembler directive. Then, assign `Dstart` the value of "." in the SECTIONS directive as follows:

```
SECTIONS
{
     .text: {}
     .data: { Dstart = .; }
     .bss : {}
}
```

This defines the symbol `Dstart` to be the ultimate linked address of the .data section. All references to this symbol in the program will then be correctly relocated to refer to this address.

A special type of assignment assigns a value to the "." symbol. This has the effect of adjusting the program counter within an output section and creating a hole between two input sections. Any value assigned to "." to create a hole is assumed to be relative to the beginning of the section and not the address actually represented by ".". Assignments to "." and holes are described in Section 9.10.

9.9.3 Assignment Expressions

These rules apply to linker expressions:

- Expressions can contain global symbols, constants, and the C language operators listed in Table 9-2.
- All numbers are treated as long (32-bit) integers.
- Constants are identified 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). 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 assigned the value of an absolute expression, the symbol is absolute.

The linker supports the C language operators listed in Table 9-2 in order of precedence. Operators in the same group have the same precedence.

Besides the operators listed in Table 9-2, 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 PC within the current section on the next 16-byte boundary. Since the align operator is a function of the current PC, it can only be used in the same context as "." – that is, within a SECTIONS directive.

Table 9-2. Operators in Assignment Expressions

Group 1 (Highest Precedence)		Group 6	
! ~ -	Logical Not Bitwise Not Negative	&	Bitwise AND
Group 2		**Group 7**	
* / %	Multiplication Division Mod	|	Bitwise OR
Group 3		**Group 8**	
+ -	Addition Minus	&&	Logical AND
Group 4		**Group 9**	
>> <<	Arithmetic right shift Arithmetic left shift	||	Logical OR
Group 5		**Group 10 (Lowest Precedence)**	
== ! = > < <= >=	Equal to Not equal to Greater than Less than Less than or equal to Greater than or equal to	= + = - = * = / =	Assignment A+=B → A=A+B A-=B → A=A-B A*=B → A=A*B A/=B → A=A/B

9.9.4 Symbols Defined by the Linker

The linker automatically defines three symbols than can be used by a program to determine at run time where a section has been linked. These symbols are called `etext`, `edata`, and `end`. These symbols are external, so they appear in the link map. They can be accessed in any assembly language module if they are declared with a .global assembler directive.

Values are assigned to these symbols as follows:

etext is assigned the first address following the .text output section. (It marks the end of executable code.)

edata is assigned the first address following the .data output section. (It marks the end of initialized data tables.)

end is assigned the first address following the .bss output section. (It marks the end of uninitialized data.)

9.10 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, .bss sections can also be treated as holes. This section describes how the linker handles such holes and how you can fill holes (and .bss sections) with a specified value.

9.10.1 Initialized and Uninitialized Sections

There are two rules to remember about the contents of an output section. An output section will contain:

Rule 1 Raw data for the *entire* section **or**
Rule 2 *No* raw data (uninitialized)

A section that has raw data is referred to as initialized. This means that the object file contains the actual memory image contents of the section. When the section is loaded, this image is loaded into memory at the section's specified starting address. The .text and .data sections **always** have raw data if anything was assembled into them. All named sections (defined with the .sect assembler directive) also have raw data in the object file.

By default, the .bss section has no raw data (it is uninitialized). It simply occupies space in the memory map, but has no actual contents. This type of section is typically used to 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. However, no memory image is stored in the file.

9.10.2 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 when building an output section. When such a hole is created, *the linker must follow rule 1 (above) and supply raw data for the hole.*

Holes can only be created *within* output sections. There can also be space *between* output sections, but such spaces are not considered to be holes. Space between output sections cannot be filled or initialized.

To create a hole in an output section, you must use a special type of linker assignment statement within a SECTIONS definition. The assignment statement modifies the SPC (denoted by the "." symbol), by either adding to it, assigning a new (greater) value to it, or aligning it at an address boundary. The operators, expressions, and syntax of assignment statements are described in Section 9.9 (page 9-27).

The following example shows how holes can be created in output sections using assignment statements:

```
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
   }
}
```

In this example, 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 that aligns the SPC on a 16-byte boundary.
- Finally, the .text section from `file3.obj` is included.

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 that aligns "." in the preceding example, which aligns .text in `file3.obj` to start on a 16-byte boundary within `outsect`. If `outsect` is ultimately allocated to start on an address that is not aligned, then .text in `file3` will also not be aligned. Assignments and alignments are relative to the beginning of the section.

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.

Another way to create a hole in an output section is to combine an uninitialized section (.bss) with initialized sections to form a single output section. *In this case, the linker treats the .bss section as a hole and supplies data for it.* An example of creating a hole in this way is:

```
SECTIONS
{
    outsect:
    {
        file1.obj(.text)
        file1.obj(.bss)       /* This becomes a hole */
    }
}
```

Since the .text section has raw data, all of `outsect` must also contain raw data (rule 1). Therefore, the uninitialized .bss section becomes a hole.

Note that uninitialized (.bss) sections only become holes when they are combined with initialized sections. If multiple .bss sections are linked together, and all are uninitialized, the resulting output section will also be uninitialized.

9.10.3 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 2-byte fill value that is replicated through memory until it fills the hole. The linker determines the fill value as follows:

1) If the hole was formed by combining a .bss section with an initialized section, you can specify a fill value for that specific .bss section. The value is specified with an = symbol and a 2-byte constant following the section name within a SECTIONS directive. 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 supplying the fill value after the section definition. For example,

```
SECTIONS
{
    outsect:
    {
        . += 10h;              /* This creates a hole          */
        file1.obj(.text)
        file1.obj(.bss)        /* This creates another hole    */
    } = 0FF00h                 /* This fills both holes with */
}                              /* 0FF00h                       */
```

3) If no explicit initialization is specified for a hole, the hole is filled with the value specified with the -f linker option. For example, suppose the command file `link.cmd` contains the following SECTIONS directive:

```
SECTIONS
{
    .text: { .= 100; }  /* Create a 100-byte hole */
}
```

Now invoke the linker with the -f option:

```
lnk370 -f 0FFFFh link.cmd
```

This fills the hole with 0FFFFh.

4) If no -f option is specified when the linker is invoked, then holes are filled with 0s.

Whenever a hole is created and filled in an initialized output section, the hole is identified in the link map (use the -m linker option to produce a map file) along with the value the linker used to fill it.

9.10.4 Explicit Initialization of .bss Sections

A .bss section only becomes a hole when it is combined with an initialized section. When .bss sections are combined with each other, the resulting output section is still uninitialized and has no raw data in the output file.

However, you can force an uninitialized section to be initialized simply 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: {} = 1234h     /* Fills .bss with 1234h */
}
```

> **Note:**
>
> Since 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 .bss sections or holes.

9.10.5 Examples of Using Initialized Holes

The TMS370 has 4K bytes of program memory starting at location 7000h. 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 will begin at address 7000h. Suppose also that you have a section of initialized interrupt vectors called `int_vecs` that you want to link at address 7FF0h. You could fill the space between the end of the .text section and the beginning of the interrupt vectors; this example fills the space with a 1-byte fill value of 0EFh (a trap instruction). Figure 9-3 illustrates the desired memory map for program memory.

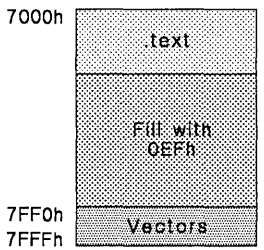


Figure 9-3. Initialized Hole

Remember, you cannot fill the space between two output sections. To obtain the configuration shown in Figure 9-3, you must create one large output section that has .text at the beginning, int_vecs at the end, and a hole filled with 0EFh in between:

```
SECTIONS
{
   prog 07000h :              /* Define prog and bind it to start at 7000h */
   {
      file1.obj(.text)        /* Link in the .text sections from each file */
      file2.obj(.text)
      file3.obj(.text)
      . = 0FF0h;              /* Create a hole up to 0FF0h (7FF0h absolute */
      file1.obj(int_vecs)     /* Link in the vectors section               */
   } = 0EFEFh                 /* Specify a fill value                      */
}
```

The fill value must be a 2-byte constant. To have the value 0EFh in each byte, the fill value was specified as 0EFEFh.

Notice that the value 0FF0h, which is assigned to the section program counter (.), is relative to beginning of the section. Since the section begins at 7000h, the hole is actually created from the end of the .text section to address 7FF0h.

9.11 Partial (Incremental) Linking

An output file that has been linked can be linked again with additional modules. This is known as incremental or **partial** linking. Partial linking allows you to partition large applications, link each part separately, and then link all the parts together to create a final executable program.

Follow these guidelines for producing a file that will be relinked:

- Intermediate files **must** have relocation information. Use the -r option when you invoke the linker to 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, or it will be removed.
- Intermediate link steps should only be concerned with the formation of output sections, and not with allocation. All allocation, binding, and MEMORY directives should be performed in the final link.

The following example shows how you can build a program using incremental linking:

Step 1: Link the file `file1.com`; use the -r option to retain relocation information in the output file `tempout1.obj`.

```
lnk370 -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.obj`.

```
lnk370 -r -o tempout2 file2.com
```

`file2.com` contains:

```
SECTIONS
{
   ss2:{
        g1.obj
        g2.obj
          .
          .
          .
        gn.obj
       }
}
```

Step 3: Link `tempout1.obj` and `tempout2.obj`:

```
lnk370 -m final.map -o final.out tempout1 tempout2
```

9.12 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:
0002 to 0100	Register file
1F00 to 1FFF	Data EEPROM
2000 to 3FFF	8K external RAM
7000 to 7FFF	4K internal program ROM

The program will be built from the following elements:

- Register variables, declared with the .reg assembler directive, are in the .reg sections of `demo.obj` and `ctrl.obj`.
- 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 7FF0h in program ROM.
- A table of coefficients, contained in the .data sections of `tables.obj` and `ctrl.obj`, must be linked into EEPROM. The remainder of EEPROM must be initialized with the value 0A26E.
- A set of variables, contained in the .bss section of `ctrl.obj` must be linked into the register file along with the .reg sections. These variables must be preinitialized to 0FFh.
- Another .bss section in `demo.obj` must be linked into external RAM.

Figure 9-4 illustrates the linker command file for this example; Figure 9-5 illustrates the map file.

```
/*************************************************************************/
/****                    Specify 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
{
    RFILE   :  origin = 0002h     length = 00FEh
    EEPROM  :  origin = 1F00h     length =  100h
    RAM     :  origin = 2000h     length = 2000h
    ROM     :  origin = 7000h     length = 1000h
}

/*************************************************************************/
/****                  Specify the Output Sections                    ****/
/*************************************************************************/

SECTIONS
{
    .reg:  {}  > RFILE          /* Link relocatable registers          */

    .text: {}  > ROM            /* Link all .text sections into ROM    */

    int_vecs 7FF0h: {}          /* Link interrupts at 7FF0h            */

    .data:                      /* Link the .data sections             */
    {
        tables.obj(.data)
        . += 100;               /* Create a hole to end of the block   */
    } = 0A26Eh > EEPROM         /* Fill and link into EEPROM           */

    ctrl_vars:                  /* Create a new section for ctrl vars  */
    {
        ctrl.obj(.bss)
    } = 0FFFFh > RFILE          /* Fill with 0FFFFh and link to RFILE  */

    .bss: {}   >RAM             /* Link all remaining .bss sections    */
}

/*************************************************************************/
/****                      End of Command File                        ****/
/*************************************************************************/
```

Figure 9-4. Linker Command File, demo.cmd

Now invoke the linker by entering the following command:

```
lnk370 demo.cmd
```

This creates the map file shown in Figure 9-5 and an output file called demo.out that can be run on the TMS370.

```
*****************************************************************
370 COFF Linker, Version 1.04,85.319
*****************************************************************

OUTPUT FILE NAME:   <demo.out>
ENTRY POINT SYMBOL: "SETUP"  address:  00007000


MEMORY CONFIGURATION
          name      origin          length          attributes
        --------   --------        ---------        ----------
        RFILE      00000002        0000000FE           RWIX
        EEPROM     00001F00        000000100           RWIX
        RAM        00002000        000002000           RWIX
        ROM        00007000        000001000           RWIX


SECTION ALLOCATION MAP

 output                                          attributes/
section  page    origin        length            input sections
-------  ----  ----------    ----------      ----------------
.reg      0    00000002      00000047        UNINITIALIZED
                 00000002        00000021      demo.obj (.reg)
                 00000023        00000026      ctrl.obj (.reg)

ctrl_var  0    00000049      0000002A        ctrl.obj (.bss) [fill = ffff]
                 00000049        0000002A

.data     0    00001F00      00000100
                 00001F00        000000A5      tables.obj (.data)
                 00001FA5        0000005B      --HOLE-- [fill = A26E]

.bss      0    00002000      0000009A        UNINITIALIZED
                 00002000        0000009A      demo.obj (.bss)

.text     0    00007000      000001B2
                 00007000        0000014E      demo.obj (.text)
                 0000714E        00000064      ctrl.obj (.text)

int_vec   0    00007FF0      000000010
                 00007FF0        000000010     tables.obj (int_vecs)


GLOBAL SYMBOLS
 address    name                                 address    name
---------  ----                                 ---------  ----
00007000    SETUP                               0000000E    aper
00000022    amp                                 0000001C    time
0000000E    aper                                00000022    amp
0000704A    ctrl                                00002000    edata
00002000    edata                               00002000    extvar
0000209A    end                                 0000209A    end
000071B2    etext                               00007000    SETUP
00002000    extvar                              0000700A    start
0000700A    start                               0000704A    ctrl
0000001C    time                                000071B2    etext

[10 symbols]
```

Figure 9-5. Output Map File, demo.map

10. Absolute Lister Description

The TMS370 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.

Topics in this section include:

10.1 Producing an Absolute Listing

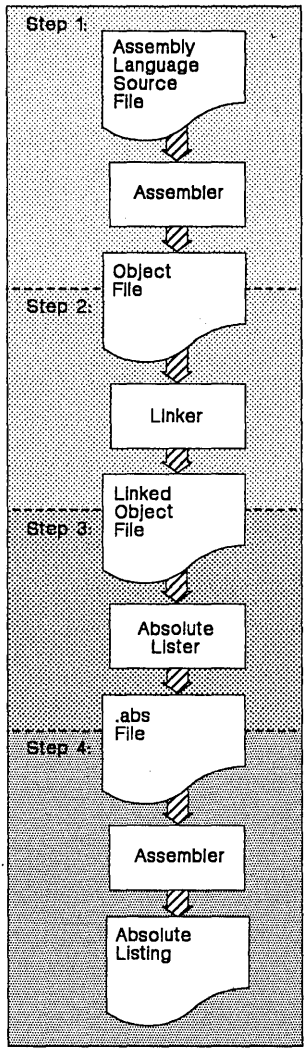


Figure 10-1. Absolute Lister Development Flow

Figure 10-1 illustrates the steps required to produce an absolute listing.

1) First, assemble a source file.
2) Link the resulting object file.
3) Invoke the absolute lister; use the linked object file as input. This creates a file with a .abs extension.
4) Finally, assemble the .abs file; you must invoke the assembler with the -a option. This produces a listing file that contains absolute addresses.

10.2 Invoking the Absolute Lister

There are two methods for invoking the absolute lister:

- **Method 1:**

  1) Invoke the absolute lister with the following command:

     ```
     abs370    [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 will prompt you for a filename.

  2) 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**.

     To create the absolute listing, you must assemble this file and use the -a assembler option:

     ```
     asm370 filename.abs -a
     ```

- **Method 2**

  When you want to produce absolute listings of several files, you can use a batch file called `abs.bat` that is included with the assembly language tools package. This file invokes the absolute lister, but allows you to specify as many filenames as you wish:

  ```
  abs <file 1> <file 2> [ ... <file n>]
  ```

  These files **must** have been successfully linked together in the `a.out` file that is produced by the linker. Do not specify extensions for the filenames when you invoke the absolute lister in this way.

  The abs.bat file automatically invokes the assembler; thus, abs.bat creates absolute listing files with the .lst extension.

  You can edit the `abs.bat` file to customize it. For example, the `abs.bat` file specifies that the filenames are linked in the file `a.out`; you can change that to another linker output file.

10.3 Absolute Lister Examples

This example uses three source files. Note that `module1.asm` and `module2.asm` both include the file `globals.def`.

module1.asm

```
.include "globals.def"
.reg     xflags,2
.reg     flags
.text
sbit0    Gflag
```

module2.asm

```
.include "globals.def"
.text
.sbit1   Gflag
```

globals.def

```
        .globreg flags
Gflag   .dbit    3,flags
```

The following steps will create absolute listings for the files `module1.asm` and `module2.asm`:

1) First, assemble `module1.asm` and `module2.asm`:

```
asm370 module1
asm370 module2
```

This creates two object files called `module1.obj` and `module2.obj`.

2) Next, link `module1.obj` and `module2.obj`. We'll use the following linker command file, called `bittest.lnk`.

```
/*****************************************************/
/*  File bittest.lnk -- COFF linker control file     */
/*  for linking TMS3370 modules                      */
/*****************************************************/

-o BITTEST.OUT        /* executable output filename */
-m BITTEST.MAP        /* output map file            */

/* input files */

MODULE1.OBJ
MODULE2.OBJ

/* define 370 memory map */

MEMORY
{
   RFILE:   origin=0002h    length=00FEh
   PFILE:   origin=1000h    length=0100h
   EEPROM:  origin=1F00h    length=0100h
   ROM:     origin=7000h    length=1000h
   XROM:    origin=8000h    length=4000h
}

/* define the output sections */

SECTIONS
{
   .reg:          {}  >RFILE
   .data:         {}  >EEPROM
   .text          {}  >ROM
}
```

Invoke the linker:

```
lnk370 bittest.lnk
```

This creates an executable object file called `bittest.out`; we'll use this new file as input for the absolute lister.

3) Now, invoke the absolute lister:

```
abs370 bittest.out
```

This will create two files called `module1.abs` and `module2.abs`:

module1.abs:

```
flags       .setsym         04h
            .setsect        ".text"       , 07000h
            .setsect        ".data"       , 01F00h
            .setsect        ".bss"        , 05h
            .setsect        ".reg"        , 02h
            .text
            .include        "module1.asm"
```

module2.abs:

```
flags       .setsym         04h
            .setsect        ".text"       , 07003h
            .setsect        ".data"       , 01F00h
            .setsect        ".bss"        , 05h
            .text
            .include        "module2.asm"
```

These abs files have information that the assembler needs when you invoke it in step 4:

a) The .abs files contain *.setsym* directives, which are equates for global symbols. Both these .abs 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`.

b) The .abs files contain *.setsect* directives, which define the absolute addresses for sections.

c) The .abs files contain *.include* 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 only useful for creating absolute listings.

4) Finally, assemble the .abs files created by the absolute lister (remember that you must use the -a option when you invoke the assembler):

```
asm370  -a  module1.abs
asm370  -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 that we've created are shown on page 10-6.

module1.lst:

```
module1.abs TMS370 Assembler Version 3.00 Tue May 5 09:02:45 1987 Pg 1

  0000              0001 flags    .setsym   04h
  0000              0002          .setsect  ".text"      ,  07000h
  7000              0003          .setsect  ".data"      ,  01F00h
  1F00              0004          .setsect  ".bss"       ,  05h
  0005              0005          .setsect  ".reg"       ,  02h
  0002              0006          .text
  7000              0007          .include  "module1.asm"
  7000             A0001          .include  "globals.def"
  7000              B0001          .globreg  flags
  7000              B0002 Gflag    .dbit     3,flags
  0002              A0002          .reg      xflags,2
  0004              A0003          .reg      flags
  7000              A0004          .text
  7000 73F704       A0005          sbit0     Gflag

There were no errors detected

module1.abs TMS370 Assembler Version 3.00 Tue May 5 09:02:45 1987 Pg 2

**************** SOURCE FILES ****************
      ID   FILENAME

           module1.abs
       A   module1.asm
       B   GLOBALS.DEF
```

module2.lst:

```
module2.abs TMS370 Assembler Version 3.00 Tue May 5 09:02:59 1987 Pg 1

  0000              0001 flags    .setsym   04h
  0000              0002          .setsect  ".text"      ,  07003h
  7003              0003          .setsect  ".data"      ,  01F00h
  1F00              0004          .setsect  ".bss"       ,  05h
  0005              0005          .text
  7003              0006          .include  "module2.asm"
  7003             A0001          .include  "globals.def"
  7003              B0001          .globreg  flags
  7003              B0002 Gflag    .dbit     3,flags
  7003              A0002          .text
  7003 740804       A0003          sbit1     Gflag

There were no errors detected

module2.abs TMS370 Assembler Version 3.00 Tue May 5 09:02:59 1987 Pg 2

**************** SOURCE FILES ****************
      ID   FILENAME

           module2.abs
       A   module2.asm
       B   GLOBALS.DEF
```

11. Code Conversion Utility Description

Most EPROM programmers do not accept COFF object files as input. The code conversion utility converts a COFF object file into one an object format that most EPROM programmers will accept as input; it converts it into Intel hex object format.

The code conversion utility accepts one COFF object file as input, and produces one output file.

This section contains the following topics:

Figure 11-1 illustrates the code conversion utility's role in the assembly language development process.

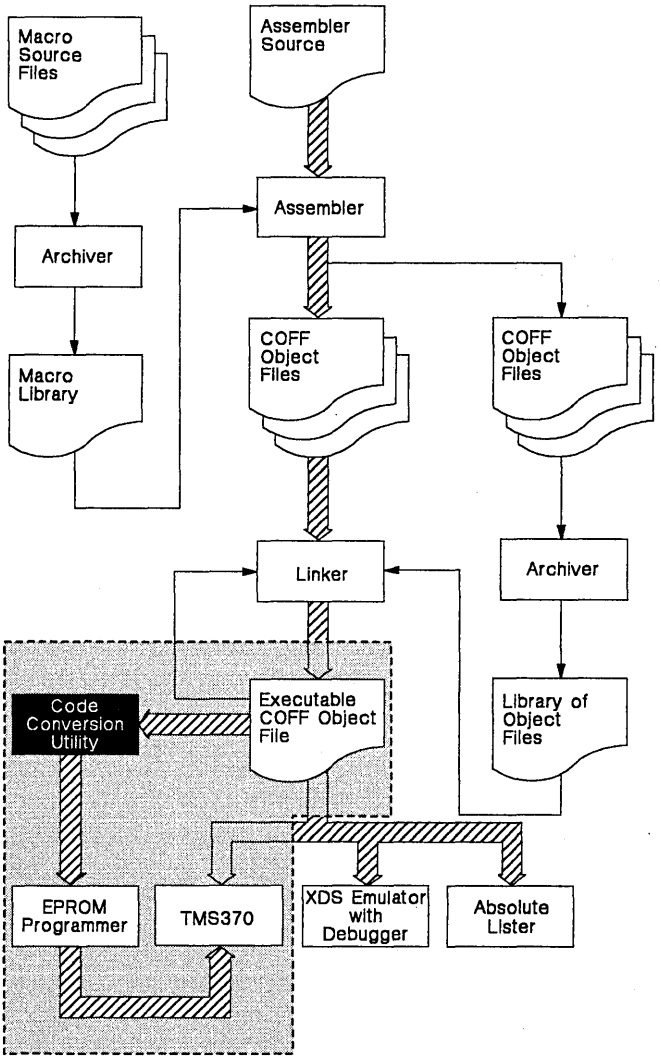


Figure 11-1. Code Conversion Utility Development Flow

11.1 Invoking the Code Conversion Utility

To invoke the code conversion utility, enter:

```
CFTOIN  [<COFF input file>] [<Intel hex output file>]
```

All the parameters are optional, but their order is significant; the first and second filenames (if used) are interpreted as the input file and the output file, respectively.

If you do not specify an input filename, the code conversion utility will prompt you for it. If you specify a filename without an extension, the utility will assume that the input filename has a default extension of **.out**.

If you do not specify a filename for the output file, the code conversion utility will prompt you for one. There is no default filename or extension for the output file.

When the utility finishes converting the input file, it will print the message `Translation complete`.

11.2 Code Conversion Utility Examples

Here are some examples of using the code conversion utility.

- **Example 1:**

  You can enter the names of the input file and the output file on the command line:

  ```
  CFTOIN file.tmp file.in
  ```

  The code conversion utility will use `file.tmp` as the input file, and place the Intel hex format output into the file `file.in`.

- **Example 2:**

  You can invoke the code conversion utility with no parameters. The utility will print the following prompts:

  ```
  COFF Input File :
  Output File :
  ```

  If, for example, you responded to the first prompt with a filename of `graphic`, the code conversion utility would use the file `graphic.out` as an input file. If you responded to the second prompt with a filename of `iout`, the code conversion utility would use the file `iout` as the output file (notice that the utility will not supply a default extension for the output file).

- **Example 3:**

  You can invoke the code conversion utility with one parameter:

  ```
  CFTOIN temp1
  ```

  The utility will use `temp1.out` as the input file, and then prompt for an output filename:

  ```
  Output File :
  ```

A. Common Object File Format

The TMS370 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 was chosen 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*. Section 3, Introduction to Common Object File Format, discusses COFF sections in detail. If you understand section theory, you will be able to use the TMS370 assembly language tools more efficiently.

This appendix contains technical details about COFF object file structure. Most of this information pertains to the symbolic debugging information that is produced by a C compiler. The main purpose of this appendix is to provide supplementary information for those of you who are interested in symbolic debugging.

Topics in this appendix include:

A.1 File Structure

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 section (except .bss),
- Relocation information for each section (except .bss),
- Line number entries for each section (except .bss),
- A symbol table, **and**
- 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 will not contain relocation entries. Figure A-1 illustrates the overall object file structure.

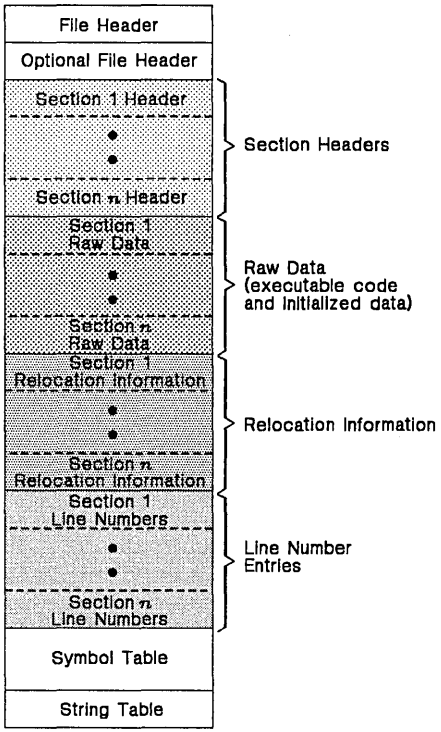


Figure A-1. COFF File Structure

Figure A-2 shows a typical example of a COFF object file that contain the three default sections, .text, .data, .bss, .reg, and a named section (referred to as <named>). By default, the .text, .data, .bss, and .reg sections, respectively, are placed in the object file, followed by any named sections in the order in which they were encountered. Although the .bss section has a section header, notice that there is no raw data, relocation information , or line number entries for .bss. This is because the .bss directive simply reserves space for uninitialized data; the .bss section contains no actual code.

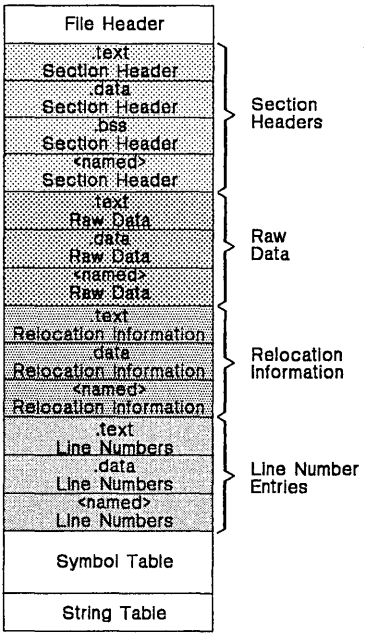


Figure A-2. Sample COFF Object File

A.2 File Header

The file header contains 20 bytes of information that describe the general format of an object file. Table A-1 shows the structure of the file header.

Table A-1. File Header Contents

Byte Number	Type	Description
0–1	Unsigned short integer	Magic number (091h) that indicates that the file can be executed in a TMS370 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, then there is no optional file header
18–19	Unsigned short integer	Flags (see Table A-2)

Table A-2 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, then F_RELFLG and F_EXEC are both set.)

Table A-2. File Header Flags (Bytes 18 and 19)

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	0010h	Local symbols were stripped from the file
F_QR32WR	0040h	The file has the byte ordering used by the TMS370 (16 bits per word, least significant byte first)

A.3 Optional File Header

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. Table A-3 illustrates the optional file header format.

Table A-3. Optional File Header Contents

Byte Number	Type	Description
0–1	Short integer	Magic number (0108h)
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 bytes
12–15	Long integer	Size (in bytes) of uninitialized data
16–19	Long integer	Beginning address of executable code
24–27	Long integer	Beginning address of initialized data

A.4 Section Headers

COFF object files contain a table of section headers that specify where each section begins in the object file. Each section of the file has its own section header. A section is padded so that its size is a multiple of two bytes.

Table A-4. Section Header Contents

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 bytes
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	Character	Memory page number

Table A-5 lists the flags that can appear in bytes 36 and 37 of the section header.

Table A-5. Section Header Flags (Bytes 36 and 37)

Mnemonic	Flag	Description
STYP_REG	0000h	Regular section (allocated, relocated, loaded)
STYP_DSECT	0001h	Dummy section (not allocated, relocated, not loaded)
STYP_NOLOAD	02h	Noload section (allocated, relocated, not loaded)
STYP_GROUP	0004h	Grouped section (formed from several input sections)
STYP_PAD	0008h	Padding section (not allocated, not relocated, loaded)
STYP_COPY	0010h	Copy section (for a decision function in updating fields; not allocated, relocated, loaded; 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
STYP_ALIGN	0100h	Section is aligned on a cache boundary

Note: The term *loaded* means that the raw data for this section appears in the object file.

The flags listed in Table A-5 can be combined; for example, if bytes 36 and 37 are set to 024h, then STYP_GROUP and STYP_TEXT are both set.

Figure A-6 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.

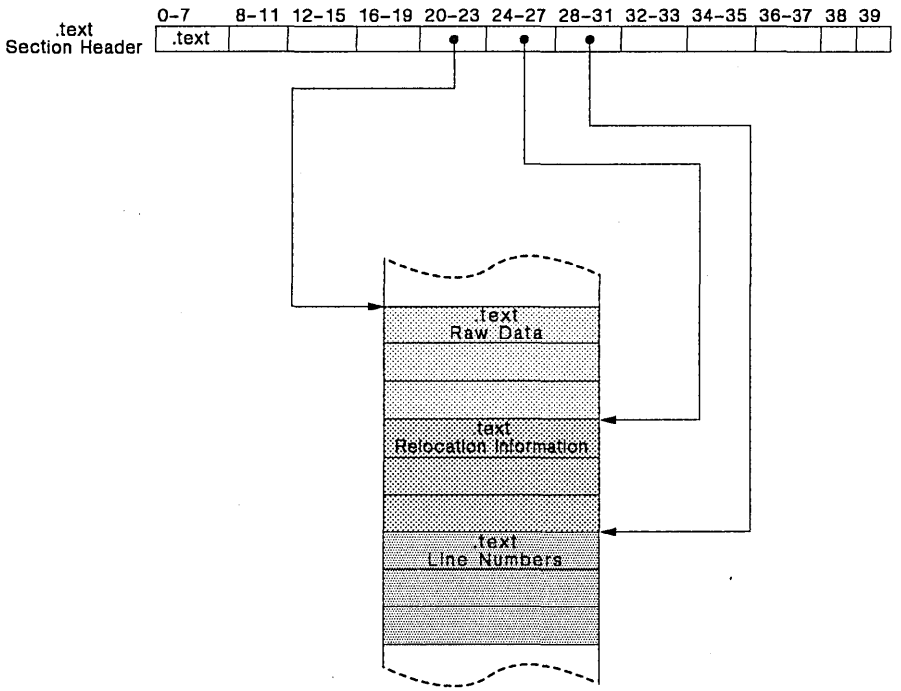


Figure A-3. An Example of Section Header Pointers for the .text Section

As Figure A-2 (page A-3) shows, the .bss section varies from this format. Although there is a section header for the .bss section, the .bss section has no raw data, no relocation information, no line number information, and occupies no actual space in the object file. Therefore, the number of relocation entries, the number of line number entries, and the file pointers in a .bss section header are zero. The .bss section header simply tells the linker how much space for variables it should reserve in the memory map.

A.5 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 the input section are treated.

The relocation information entries use the 10-byte format shown in Table A-6.

Table A-6. Relocation Entry Contents

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 A-7 lists the relocation types that can appear in bytes 8 and 9 of the relocation entry.

Table A-7. Relocation Types (Bytes 8 and 9)

Mnemonic	Flag	Relocation Type
R_RELBYTE	0Fh	8-bit direct (relocatable register)
R_RELWORD	10h	16-bit direct

A.6 Line Number Table

The object file contains a table of line number entries that are useful for symbolic debugging. When a 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. Table A-8 shows the format of a line entry.

Table A-8. Line Number Entry Format

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, then this is the first line of a function entry 2) If this field is *not* 0, then this is the line number of a line of C source code

The line entry table can contain many of these blocks.

Symbol Index	0
physical address	line number
physical address	line number
.	.
.	.
.	.
Symbol Index	**0**
physical address	line number
physical address	line number

Figure A-4. Line Number Blocks

As Figure A-4 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 words 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.

Figure A-9 illustrates an example of 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 words, the second line produces 3 words, and the third line produces 10 words. Figure A-9 shows what the line number entries would look like for this example.

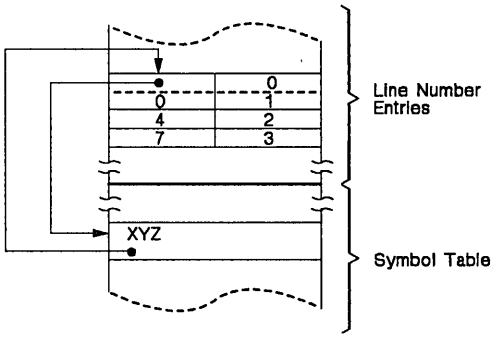


Figure A-5. Line Number Entries Example

(Note that the symbol table entry for XYZ has a field that points back to the beginning of the line number block.)

Since line numbers are not often needed, the linker provides an option (-s) that strips line number information from the object file, providing a more compact object module.

A.7 Symbol Table

The order of symbols in the symbol table is very important; they appear in the sequence shown in Figure A-3.

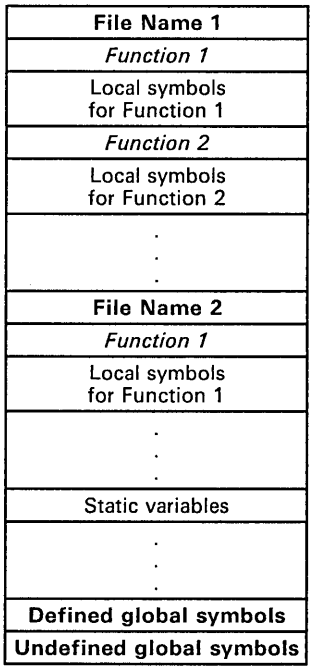


Figure A-6. Symbol Table Contents

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 a pointer 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 Table A-9. Some symbols may not have all the characteristics listed above; if a particular field is not set, it will be set to null.

Table A-9. Symbol Table Entry Contents

Byte Number	Type	Description
0–7	Character	This field contains one of the following: 1) An 8-character symbol name, padded with nulls 2) A pointer 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)

A.7.1 Special Symbols

The symbol table contains some special symbols that are generated by the compiler, assembler, and linker. Table A-10 lists these symbols.

Table A-10. Special Symbols in the Symbol Table

Symbol	Description
.file	File name
.text	Address of .text section
.data	Address of .data section
.bss	Address of .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, etext	Next available address after the end of the .text output section
—edata, edata	Next available address after the end of the .data output section
—end, end	Next available address after the end of the .bss output section

Several of these symbols appear in pairs:

- .bb/.eb indicate the beginning and ending of a block.
- .bf/.ef indicate the beginning and ending 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.

Each special symbol contains ordinary symbol table information as well as an auxiliary entry.

A.7.1.1 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. Figure A-7 shows how block symbols are grouped in the symbol table.

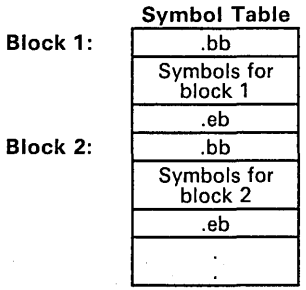


Figure A-7. Symbols for Blocks

A.7.1.2 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 name of the function precedes the .bf special symbol. Figure A-8 shows the format of symbol table entries for a function.

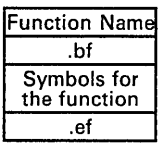


Figure A-8. Symbols for Functions

If a function returns a structure or union, then a symbol table entry for the special symbol .target will appear between the entries for the function name and the .bf special symbol.

A.7.2 Symbol Names

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, than 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, then 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.

A.7.3 String Table

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 instead contains 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.

Figure A-5 shows an example of a string table that contains two symbol names, Rotation_Coordinate and Shade_Pattern. The index in the string table is 4 for Rotation_Coordinate and 24 for the Shade_Pattern.

38			
'R'	'o'	't'	'a'
't'	'i'	'o'	'n'
'_'	'C'	'o'	'o'
'r'	'd'	'i'	'n'
'a'	't'	'e'	'\0'
'S'	'h'	'a'	'd'
'e'	'_'	'P'	'a'
't'	't'	'e'	'r'
'n'	'\0'		

Figure A-9. Sample String Table

A.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 a C compiler accesses a symbol. Table A-11 lists valid storage classes.

Table A-11. Symbol Storage Classes

Mnemonic	Value	Storage Class	Mnemonic	Value	Storage Class
C_NULL	0	No storage class	C_UNTAG	12	Union tag
C_AUTO	1	Automatic variable	C_TPDEF	13	Type definition
C_EXT	2	External symbol	C_USTATIC	14	Uninitialized static
C_STAT	3	Static	C_ENTAG	15	Enumeration tag
C_REG	4	Register variable	C_MOE	16	Member of an enumeration
C_EXTDEF	5	External definition	C_REGPARM	17	Register parameter
C_LABEL	6	Label	C_FIELD	18	Bit field
C_ULABEL	7	Undefined label	C_BLOCK	100	Beginning or end of a block; used only for the .bb and .eb special symbols
C_MOS	8	Member of a structure	C_FCN	101	Beginning or end of a function; used only for the .bf and .ef special symbols
C_ARG	9	Function argument	C_EOS	102	End of structure; used only for the .eos special symbol
C_STRTAG	10	Structure tag	C_FILE	103	Filename; used only for the .file special symbol
C_MOU	11	Member of a union	C_LINE	104	Used only by utility programs

Some special symbols are restricted to certain storage classes. Table A-12 lists these symbols and their storage classes.

Table A-12. Special Symbols and Their Storage Classes

Special Symbol	Restricted to this Storage Class
.file	C_FILE
.bb	C_BLOCK
.eb	C_BLOCK
.bf	C_FCN
.ef	C_FCN
.eos	C_EOS
.text	C_STAT
.data	C_STAT
.bss	C_STAT

A.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; Table A-13 summarizes the storage classes and related values.

Table A-13. Symbol Values and Storage Classes

Storage Class	Value Description
C_AUTO	Stack offset in bits
C_EXT	Relocatable address
C_STAT	Relocatable address
C_REG	Register number
C_LABEL	Relocatable address
C_MOS	Offset in bits
C_ARG	Stack offset in bits
C_STRTAG	0
C_MOU	Offset in bits
C_UNTAG	0
C_TPDEF	0
C_ENTAG	0
C_MOE	Enumeration value
C_REGPARM	Register number
C_FIELD	Bit displacement
C_BLOCK	Relocatable address
C_FCN	Relocatable address
C_FILE	0

If a symbol's storage class is C_FILE, then 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.

A.7.6 Section Number

Bytes 12–13 of a symbol table entry contain a number that indicates which section the symbol was defined in. Table A-14 lists these numbers and the sections they indicate.

Table A-14. Section Numbers

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
N_SCNUM	2	.data section
N_SCNUM	3	.bss section
N_SCNUM	4–77,777	Section number of a named section, in the order in which the named sections are encountered

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, then 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.

A.7.7 Type Entry

Bytes 14–15 of the symbol table entry define the symbol's type. Each symbol symbol has:

- **One** basic type
- One to six derived types

The format for this 16-bit type entry is:

	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. Table A-15 lists valid basic types.

Table A-15. Basic Types

Mnemonic	Value	Type
T_NULL	0	Type not assigned
T_CHAR	2	Character
T_SHORT	3	Short integer
T_INT	4	Integer
T_LONG	5	Long integer
T_FLOAT	6	Floating point
T_DOUBLE	7	Double word
T_STRUCT	8	Structure
T_UNION	9	Union
T_ENUM	10	Enumeration
T_MOE	11	Member of an enumeration
T_UCHAR	12	Unsigned character
T_USHORT	13	Unsigned short integer
T_UINT	14	Unsigned integer
T_ULONG	15	Unsigned long integer

Bits 4–15 of the type field are arranged as six 2-bit fields which can indicate 1 to 6 derived types. Table A-16 lists the possible derived types.

Table A-16. Derived Types

Mnemonic	Value	Type
DT_NON	0	No derived type
DT_PTR	1	Pointer
DT_FCN	2	Function
DT_ARY	3	Array

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.

A.7.8 Auxiliary Entries

Each symbol table may have a **one** or **no** auxiliary entry. An auxiliary table entry contains the same number of bytes as a symbol table entry (18), but the format of an auxiliary entry depends on its type and storage class. Table A-17 summarizes these relationships.

Table A-17. Auxiliary Symbol Table Entries Format

Name	Storage Class	Type Entry		Auxiliary Entry Format
		Derived Type 1	Basic Type	
.file	C_FILE	DT_NON	T_NULL	Filename
.text, .data, .bss	C_STAT	DT_NON	T_NULL	Section
tagname	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_NULL	Tag name
.eos	C_EOS	DT_NON	T_NULL	End of structure
fcname	C_EXT C_STAT	DT_FCN	(See note 1)	Function
arrname	(See note 2)	DT_ARY	(See note 1)	Array
.bb, .eb	C_BLOCK	DT_NON	T_NULL	Beginning and end of a block
.bf, .ef	C_FCN	DT_NON	T_NULL	Beginning and end of a function
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

Notes: 1) Any except T_MOE.
2) C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF.

In Table A-17, *tagname* refers to any symbol name (including the special symbol *.n*fake). *Fcname* and *arrname* refer to any symbol name.

Any symbol that satisfies more than one condition in Table A-17 should have a union format in its auxiliary entry. Any symbol that does not satisfy any of these conditions should not have an auxiliary entry.

A.7.8.1 File Names

Each of the auxiliary table entries for a file name contains a 14-character file name in bytes 0–13. Bytes 14–17 are unused.

Table A-18. Section Format for Auxiliary Table Entries

Byte Number	Type	Description
0–13	Character	File name
14–17	–	Unused

A.7.8.2 Sections

The auxiliary table entries for sections have the format shown in Table A-18.

Table A-19. Section Format for Auxiliary Table Entries

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)

A.7.8.3 Tag Names

Table A-20 illustrates the format of auxiliary table entries for tag names.

Table A-20. Tag Name Format for Auxiliary Table Entries

Byte Number	Type	Description
0–5	–	Unused (zero filled)
6–7	Unsigned short 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)

A.7.8.4 End of Structure

Table A-21 illustrates the format of auxiliary table entries for ends of structures.

Table A-21. End of Structure Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Long integer	Tag index
4–5	–	Unused (zero filled)
6–7	Unsigned short integer	Size of structure, union, or enumeration
8–17	–	Unused (zero filled)

A.7.8.5 Functions

Table A-22 illustrates the format of auxiliary table entries for functions.

Table A-22. Function Format for Auxiliary Table Entries

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)

A.7.8.6 Arrays

Table A-23 illustrates the format of auxiliary table entries for arrays.

Table A-23. Array Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Long integer	Tag index
4–5	Unsigned short integer	Line number declaration
6–7	Unsigned short 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)

A.7.8.7 End of Blocks and Functions

Table A-24 illustrates the format of auxiliary table entries for the ends of blocks and functions.

Table A-24. End of Blocks and Functions Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	–	Unused (zero filled)
4–5	Unsigned short integer	C source line number
6–17	–	Unused (zero filled)

A.7.8.8 Beginning of Blocks and Functions

Table A-25 illustrates the format of auxiliary table entries for the beginnings of blocks and functions.

Table A-25. Beginning of Blocks and Functions Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	–	Unused (zero filled)
4–5	Unsigned short integer	C source line number
6–11	–	Unused (zero filled)
12–15	Long integer	Index of next entry past this block
16–17	–	Unused (zero filled)

A.7.8.9 Names Related to Structures, Unions, and Enumerations

Table A-26 illustrates the format of auxiliary table entries for the names of structures, unions, and enumerations.

Table A-26. Structure, Union, and Enumeration Names Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Long integer	Tag index
4–5	–	Unused (zero filled)
6–7	Unsigned short integer	Size of the structure, union, or enumeration
8–17	–	Unused (zero filled)
16–17	–	Unused (zero filled)

B. Assembler Error Messages

During the assembly process, the assembler issues messages about the errors it encounters. These errors are printed on the screen. Within the listing file, the assembler flags the lines that produced errors with the following letter codes:

B Bit operation error
C Constant or conditional assembly error
E Invalid expression
I Instruction error
L Library or local label error
M Macro error
O Operand error
R Register error
S Syntax error
U Undefined symbol error
Z Zero operation error

Most of these errors are fatal; that is, they prevent the assembler from creating an object file.

This remainder of this section lists the assembler error messages, listed in alphabetical order according to error type.

- ***Error type B - bit operation errors***

  bit name must be defined before use: A bit name must be defined with the .dbit directive before it can be used.

  bit number out of range: Bit numbers are limited to the range 0–7.

  name of bit expected: The operand of this instruction should be a bit named with the .dbit directive.

- ***Error type C - constant errors***

  invalid constant format

- ***Error type C - conditional errors***

  else needs corresponding if An .else was specified without a preceding .if.

  unexpected endif encountered: An .endif directive has been encountered without a preceding .if directive.

- ***Error type E - invalid expression errors***

  ')' expected: A parenthetical expression was not terminated.

  attempt to redefine section address: You can only use the address parameter of a sections directive one time.

  bad type of expression

cannot redefine symbol to a register: This symbol cannot be defined as a register.

can't open include file: The file specified with the .include directive cannot be found.

end of conditional block missing: An .endif directive is needed.

end of string not found: A string must close with a " symbol.

expression not terminated properly

filename must be within quotes: A filename must be specified as a string (enclosed in double quotes).

illegal operation in expression

include file nested too deep: Include files can only be nested to a maximum of eight levels.

invalid expression

invalid listing length: The maximum length of a listing page is 100 lines.

invalid listing width: The maximum width of a listing page is 200 characters.

invalid trap number

string expected The operand must be a string.

symbol multiply defined: A symbol (or label) has been defined more than once.

too many include files: Only 26 files can be included in a single source file.

unexpected end encountered: An .end directive has been encountered within a conditional block, a repeat block, or a macro definition.

- ***Error type I - instruction errors***

  bad instruction or directive: The specified instruction or directive is not valid.

- ***Error type L - library errors***

  cannot open library: The library specified with the .mlib directive cannot be found.

  library not archive: The filename specified with the .mlib directive does not name a file that is in archive format.

- ***Error type L - local label errors***

  illegal use of local label

  local label multiply defined in block: A local label has been defined again before it was reset.

local label not defined: A jump instruction has an undefined local label as its operand.

- *Error type M - macro errors*

  bad macro definition: The macro definition does not follow correct syntax.

  .ENDM statement missing in macro: Macro definition is not ended.

  .IF level exceeded: .if directives can only be nested to a maximum of eight levels.

  include files not allowed in macro: You cannot use the .include directive in a macro definition.

  incorrect macro definition: The macro definition does not conform to correct syntax.

  invalid if structure

  invalid if/loop nesting

  invalid macro library pathname: The specified macro library name cannot be found or opened.

  invalid macro qualifier: The macro is qualified with an invalid qualifier suffix.

  invalid macro verb: The specified macro directive is not recognized as valid.

  label not defined: An instruction uses a label (as an operand) that has not been defined.

  long macro variable qualifier

  loop nesting level exceeded The .loop directives can only be nested to a maximum of eight levels.

  macro line too long

  macro nesting level exceeded

  too many macro variables: Only 128 variables can be defined in a single macro.

  variable already defined: This macro redefines a variable that has already been defined within the same macro.

- *Error type O - operand errors*

  address is illegal operand: The specified address is not legal.

  directive needs an absolute value: All expressions used as directive operands must be positive values.

  illegal relative address

  illegal use of global label

  invalid operand or operand combination

undefined expression

value cannot be external: This instruction cannot use a global symbol as an operand.

value must be positive: This instruction must have positive values as operands.

value out of range

- ***Error type R - register errors***

  cannot redefine register: A register cannot be redefined.

  illegal peripheral number: The range of peripheral registers is P0–P255.

  illegal register number: The range of registers is R0–R255.

  illegal use of A register: Register A cannot be used as an operand for this instruction.

  illegal use of B register: Register B cannot be used as an operand for this instruction.

  illegal work register

  invalid register size

  register or peripheral file expected: The operand must be a register or a peripheral register.

  relocatable registers larger than 255: Only 256 relocatable registers can be defined in the .reg section.

- ***Error type S - syntax errors***

  comma expected: A comma is needed in the operand field.

  identifier expected

  label required: The .equ directive must have a symbol in the label field.

  missing operand

  syntax error: This statement does not conform to correct syntax.

  unexpected trailing operands: More operands have been used for this instruction than are allowed.

- ***Error type U - undefined symbol errors***

  undefined symbol: A symbol that is used as an operand is not defined.

- ***Error type Z - zero operation errors***

  attempt to divide by zero: Dividing by zero is illegal.

C. Linker Error Messages

The linker issues several types of error messages:

- Corrupt input files
- Output errors
- Allocation errors
- Directives errors
- Incorrect expressions
- Options misuse
- Space misuse

This section discusses the following types of errors:

C.1 Errors Caused by Corrupt Input Files

The following error messages indicate that the input file is corrupt, nonexistent, or unreadable. Make sure that the file is in the correct directory. If the file is corrupt, try reassembling it.

Many of these errors can be categorized into the following groups. Instead of "(...)", the linker will print the name of a particular object that it is attempting to interact with.

- Can't open (...)
- Can't read (...)
- Can't seek (...)
- Fail to read (...)
- Fail to seek (...)
- Fail to skip (...)
- Seek to (...) failed

The following error messages are also caused by corrupt input files.

- File (...) has no relocation information
- File (...) is of unknown type
- Illegal relocation type (...) found in section(s) of file (...)
- Invalid archive size for file (...)
- Library (...) member has no relocation information
- Line number entry found for absolute symbol
- Relocation symbol not found: index (...), section (...), file (...)
- Relocation entries out of order in section (...) of file (...)

C.2 Output Errors

These errors occur because the linker cannot write to the output file. This usually indicates that the file system is out of space. Instead of "(...)", the linker will print the name of a particular object that it is attempting to interact with.

- Cannot complete output file (...), write error
- Fail to write (...)
- Fail to copy (...)
- I/O error on output file (...)

C.3 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 SECTION directives conflict in some way. If you are using a linker command file, check that MEMORY and SECTION directives allow enough room to ensure that no sections overlap and that no sections are being placed in unconfigured memory. Instead of "(...)", the linker will print the name of a particular object that it is attempting to interact with.

- Bond address (...) for section (...) is outside all memory on page (...)
- Bond address (...) for section (...) overlays previously allocated section
- Can't allocate output section, (...) of size (...) on page (...)
- Default allocation failed: (...) is too large
- GROUP containing section (...) is too big
- Memory types (...) and (...) on page (...) overlap
- Section (...) at address (...) overlays previously allocated section (...) at address
- Section (...), bonded at address (...), won't fit into page (...) of configured memory
- Section (...) enters unconfigured memory at address (...)
- Section (...) in file (...) is too big
- Bond address (...) incompatible with alignment for section (...)
- Can't allocate section (...) with attribute (...) on page (...)
- No owner (...) for section (...) on page (...)

  (*Invalid or nonexistent memory range.*)
- Section (...) enters unconfigure memory at address (...)

C.4 Directives Errors

These errors are caused by incorrect use of linker directives. Check the input directives for accuracy. Instead of "(...)", the linker will print the name of a particular object that it is attempting to interact with.

- Adding name (...) to multiple output sections

  (*The input section is mentioned twice in the SECTION directive.*)

- Bad attribute value in MEMORY directive: (...)

  (*An attribute must be* **R**, **W**, **X**, *or* **I**.)

- Bad flag value in SECTIONS directive, option (...)

- Bad fill value

  (*The fill value must be a 2-byte constant.*)

- Bonding excludes alignment

  (*The section will be bound at the given address regardless of the alignment of that address.*)

- Cannot align a section within GROUP - (...) not aligned

- Cannot bond a section within a GROUP

- Cannot specify an owner for sections within a GROUP

  (*The entire group is treated as one unit, so the group may be aligned or bound to an address, but the sections making up the group may not be handled individually.*)

- DSECT (...) can't be given an owner

  (*Since dummy sections do not participate in memory allocation, it is meaningless for a dummy section to be given an owner or an attribute.*)

- Section (...) not built

  (*The most likely cause of this is a syntax error in the SECTIONS directive.*)

- Semicolon required after expression

- Statement ignored

  (*Caused by a syntax error in a expression.*)

- Fill value on -f flag truncated to (...) bytes

  (*Warning.*)

- Syntax error: scanned line = (...)

- Cannot specify a page for a section within a GROUP

C.5 Errors Caused by Incorrect Expressions

These errors arise from the misuse of an input expression. Check the syntax of all expressions. Instead of "(...)", the linker will print the name of a particular object that it is attempting to interact with.

- Absolute symbol (...) being redefined

  (*An absolute symbol may not be redefined.*)

- ALIGN illegal in this context

  (*Alignment of a symbol may only be done within a SECTIONS directive.*)

- Attempt to decrement "."

- Misuse of "." symbol in assignment instruction

  (*The dot symbol cannot be used in assignment statements that are outside SECTIONS directives.*)

- Symbol (...) from file (...) being redefined

  (*A defined symbol may not be redefined in an assignment statement.*)

- Undefined symbol in expression

- Illegal operator in expression

- number (...) not a power of 2

  (*For the ALIGN operator.*)

C.6 Options Misuse Errors

Review the various options you are using and check for conflicts. Instead of "(...)", the linker will print the name of a particular object that it is attempting to interact with.

- Both -r and -s flags are set; -s flag turned off.

  (*Since the -s option strips the relocation information and -r requests a relocatable object file, these options are in conflict with each other.*)

- -o file name too large (>128 char), truncated to (string).

- Option flag does not specify a number.

- Option is invalid flag.

- -e flag does not specify a legal symbol name (...)

- -f flag does not specify a 2-byte number.

- -o flag does specify a valid file name : string.
- -c requires fill value of 0 in .cinit

  (*<val> overridden.*)
- Entry point other than _c_int00 specified

  (*For -c option only.*)
- Entry point symbol (...) undefined

C.7 Space Errors

The following errors occur if the linker attempts to allocate more space than is available in target system memory; try to decrease the amount of space used by the linker. One way to accomplish this is by making the linker command file less complex, or by using the -r option to create intermediate files.

- Internal error : aux table overflow
- Memory allocation failure
- No symbol map produced - not enough memory

C.8 Miscellaneous Errors

Instead of "(...)", the linker will print the name of a particular object that it is attempting to interact with.

- Cannot create output file (...)
- File (...) has no relocation information
- File (...) is of unknown type, magic number = (...)
- Ifile (comfile) nesting exceeded with file (...)

  (*Command file nesting is allowed up to 16 levels.*)
- section (...) not found

  (*An input section specified in a SECTIONS directive was not found in the input file.*)
- Sections .text, .data, or .bss not found

  (*Optional header may be useless.*)
- Undefined symbol (...) first referenced in file (...)

  (*Unless the -r option is used, the linker requires that all referenced symbols are defined.*)
- Unexpected EOF(End Of File)

  (*Syntax error in the linker command file.*)

- Internal symbol (...) redefined in file (...)
  (*Ignored.*)
- DSECT (...) can't be linked to an attribute
- DSECT (...) can't be given an owner
- No input files
- Could not create map file (...)
- Symbol referencing errors - (...) not built
- Errors in input - (...) not built
- Output file (...) not executable
  (*Warning.*)
- PC-relative displacement overflow at address (...) in file (...)

D. ASCII Character Set

Base		Char	Base		Char	Base		Char	Base		Char
10	16		10	16		10	16		10	16	
0	00	NULL	32	29	SP	64	40	@	96	60	‘
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	”	66	41	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

E. Glossary

absolute address: An address that is permanently assigned to a TMS370 memory location.

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 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 .equ directive.

assignment statement: A statement that assigns a value to a variable.

attribute component: Provides information about the origin and structure of a macro variable or macro symbol.

binding: A process in which you specify a distinct address for an output section or a symbol.

.bss: This is 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.

code conversion utility: A program that converts COFF object files into Intel-format object files.

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. Comment 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, based upon 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.

.data: This is 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).

executable module: An object file that has been linked and can be executed in a TMS370 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.

field: For the TMS370, a software-configurable data type whose length can be programmed to be any value in the range of 1–32 bits.

global: Describes a 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).

hole: An area between the input sections that comprise an output section which contains no actual code or data.

incremental linking: Linking files that have already been linked.

initialized section: A COFF section that contains executable code or initialized data. These sections 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.

label: A symbol which begins in column 1 of a source statement.

length component: A component of a macro variable or macro symbol that contains the number of characters that make up the string.

linker: A software tool that combines object files to form an object module that can be allocated into TMS370 system memory and executed by the TMS370.

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 TMS370 system memory.

macro: A user-defined routine that can be used as an instruction.

macro call: The process of invoking a macro.

macro definition: A block of source statements that define the name and the code that make up a macro.

macro expansion: The source statements that are substituted for the macro call and subsequently assembled.

macro library: An archive library composed of macros. Each file in the library must contain one macro; it's name must be the same as the macro name it defines, and it must have an extension of .asm.

macro variable: A variable that is valid within a macro definition or during a macro expansion.

map file: An output file created by the linker that shows the memory configuration, section composition and allocation, and symbols and the addresses at which they were defined.

memory map: A map of TMS370 target system memory space, which is partitioned into functional blocks.

mnemonic: An instruction name that the assembler translates into machine code.

model statement: Instructions or assembler directives in a macro definition that are assembled each time a macro is invoked

named section: An initialized section that is defined with a .sect directive.

object file: A file that has been assembled or linked and contains machine-language object code.

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 objet 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 section: A final, allocated section in a linked, executable module.

overlay pages: Multiple areas of physical memory that overlay each other at the same address space. TMS370 devices can map different pages into the same address space in response to hardware select signals.

partial linking: Linking a file that will be linked again.

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.

section: A relocatable block of code or data that will ultimately occupy contiguous space in the TMS370 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 (SPC): An element of the assembler that keeps track of the current location within a section; each section has its own SPC.

source file: A file that contains C code or TMS370 assembly language code that will compiled or assembled to form an object file.

string component: A copy of a string that is passed to a macro variable by a macro parameter or assigned to a macro symbol with an $ASG directive.

string table: Symbol names that are longer than 8 characters 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.

storage class: Any entry in the symbol table that indicates how a symbol should be accessed.

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.

target memory: Physical memory in a TMS370-based system into which executable object code will be loaded.

.text: This is one of the default COFF sections. The .text section is an initialized section that contains executable code. You can use the .text directive to assemble code into the .text section.

unconfigured memory: Memory that is not defined as part of the TMS370 memory map and cannot be loaded with code or data.

uninitialized section: A COFF section that reserves space in the TMS370 memory map but that has no actual contents. These sections are built up with the .bss directive.

value component: A component of a macro variable or macro symbol that specifies the value of the variable or symbol.

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.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X