TEXAS INSTRUMENTS

TMS34010 Assembly Language Tools

User's Guide

TMS34010 Assembly Language Tools 1987

1987

Graphics Products

TMS34010
Assembly Language Tools
User's Guide

TEXAS
INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to or to discontinue any semiconductor product or service identified in this publication without notice. TI advises its customers to obtain the latest version of the relevant information to verify, before placing orders, that the information being relied upon is current.

TI warrants performance of its semiconductor products to current specifications in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Unless mandated by government requirements, specific testing of all parameters of each device is not necessarily performed.

TI assumes no liability for TI applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Copyright © 1987, Texas Instruments Incorporated

Contents

Section *Page*

Illustrations

Tables

Section 1

Introduction

The TMS34010 Graphics System Processor (**GSP**) is an advanced 32-bit microprocessor optimized for graphics systems. The GSP is a member of the TMS340 family of computer graphics products from Texas Instruments. The TMS34010 is well supported by a full set of hardware and software development tools, including a C compiler, a full-speed emulator, a software simulator, and an IBM/TI-PC development board. This document discusses the software development tools that are included with the TMS34010 assembly language tools package:

- Assembler
- Archiver
- Linker
- Object format converter
- Simulator[1]

These tools can be installed on the following systems:

- **PCs:**
  - IBM-PC with PC-DOS
  - TI-PC with MS-DOS
- **VAX:**
  - VMS
  - DEC Ultrix
  - Unix System V

The TMS34010 assembly language tools create and use object files that are in common object file format, or COFF. COFF makes modular programming easier because it encourages you to think in terms of *blocks* of code and data. Object files contain separate blocks (called **sections**) of code and data that you can load into different memory spaces. You will be able to program the TMS34010 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] The simulator is available in a PC version only.

1.1 Software Development Tools Overview

Figure 1-1 shows the TMS34010 assembly language development flow. The center section of the illustration highlights the most common path; the other portions are optional.

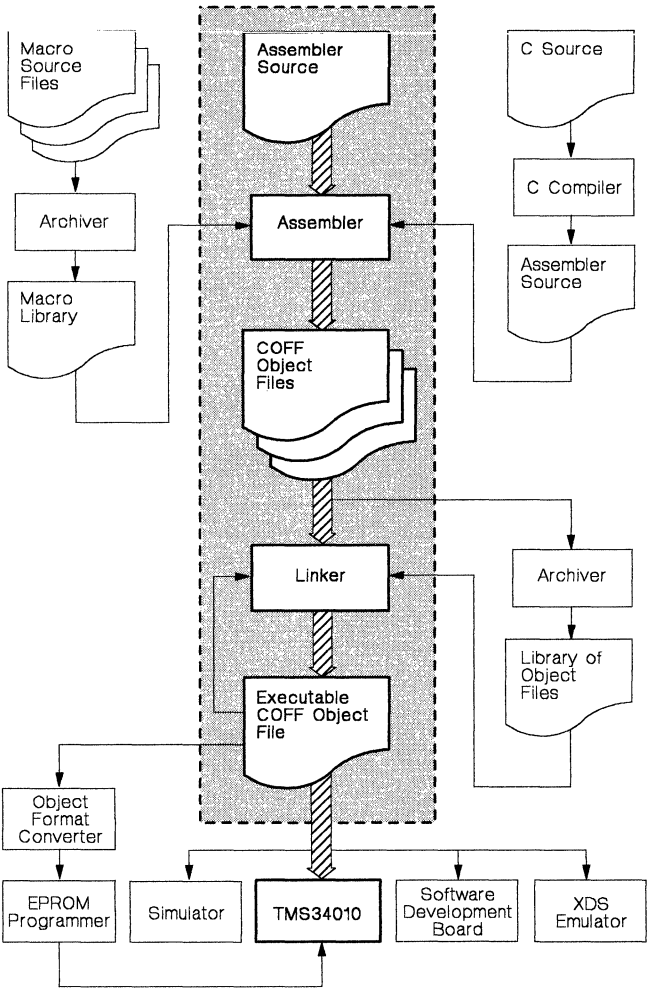


Figure 1-1. TMS34010 Assembly Language Development Flow

- The **C compiler** translates C source code into TMS34010 assembly language source code. The C compiler is not shipped as part of the assembly language tools package.

- The **assembler** translates assembly language source files into machine language object files. Source files can contain TMS34010 assembly language instructions, assembler directives, and macro directives. You can use assembler directives to control various aspects of the assembly process, such as the source listing format, data alignment, and section content.

- The **archiver** allows you to collect a group of files into a single archive file. For example, you can collect several macros together into a macro library. The assembler will search through the library and use the members that are called as macros by the source file. You can also use the archiver to collect a group of object files into an object library. The linker will include the members in the library 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 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 **TMS34010**. You can use one of several debugging tools to refine and correct your code before executing it in a TMS34010 system. These tools share similar debugger interfaces. (Note that only *linked* files can be executed.)

  - The **simulator** simulates TMS34010 functions in a configurable graphics environment. The simulator allows you to design, implement, and evaluate both graphics and nongraphics software systems. The simulator command set displays and maintains graphics and machine status information and controls execution of the software system under development.

  - The **software development board** (SDB) is a high-performance graphics card that can be used with a TI or an IBM PC. The SDB is not shipped as part of the TMS34010 assembly language package.

  - The **XDS/22 emulator** is a realtime, in-circuit emulator. It is not shipped as part of the TMS34010 assembly language package.

- Most EPROM programmers do not accept COFF object files as input. The **object format converter** converts a COFF object file into Intel hex, Tektronix hex, or TI-tagged 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 whole 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

```
        .file   "Set_Colors"
COLOR0  .set    B8
        MOVI    0FFFFh,COLOR0
COLOR1  .set    B9
        MOVI    3333h,COLOR1
```

file2.asm

```
        .file   "Set_Parms"
        EINT
CONTROL .set    0C00000B0h
        MOVI    4A4h,A1
        MOVE    A1,@CONTROL
INTENB  .set    0C0000110h
        MOVI    0C00h,A1
        MOVE    A1,@INTENB
```

2) Assemble `file1.asm`; enter:

```
gspa file1
```

The **gspa** command invokes the assembler. `file1.asm` is the input source file. (If the input file extension is .asm, you don't have to specify the extension; the assembler uses .asm as the default.) This example creates an object file called `file1.obj`. The assembler 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.

Now assemble `file2.asm`; enter:

```
gspa file2 -l
```

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

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

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

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

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

1.3 Manual Organization

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

Section 2 Software Installation
Contains instructions for installing the assembly language tools on PC and VAX systems.

Section 3 Introduction to Common Object File Format
Discusses the basic COFF concept of **sections** and how they can help you use the assembler and linker more efficiently. (Common object file format, or COFF, is the object file format that the TMS34010 assembly language tools use.) *Read Section 3 before using the assembler and linker.*

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

Section 5 Assembler Directives
Divided into two parts: the first part describes the directives according to function, the second part is a reference that presents the directives in alphabetical order.

Section 6 Instruction Set Summary
Summarizes the TMS34010 instruction set alphabetically.

Section 7 Macro Language
Describes macro directives and creating macros.

Section 8 Archiver Description
Contains instructions for invoking the archiver, creating new archive libraries, and modifying existing libraries.

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

Section 10 Object Format Converter Description
Tells you how to invoke the object format converter so that you can convert a COFF object file into an Intel or Tektronix hex object format.

Section 11 Simulator Description
Contains instructions for invoking the simulator and loading the tutorial program, explains the various fields on the simulator screen, and contains a complete alphabetical reference of simulator commands.

Appendix A Common Object File Format
Contains specific information about the internal format of COFF object files.

Appendix B Symbolic Debugging Directives

Appendix C Assembler Error Messages

Appendix D Linker Error Messages

Appendix E ASCII Character Set

Appendix F Glossary

1.4 Related Documentation

The following TMS34010 documents are also available.

- The ***TMS34010 User's Guide*** (literature number SPVU001) discusses hardware aspects of the TMS34010, such as pin functions, architecture, stack operation, and interfaces, and contains the TMS34010 instruction set. (If you received this User's Guide with the TMS34010 assembly language tools package, you should also have received a copy of the *TMS34010 User's Guide*).

- The ***TMS34010 Data Sheet*** (literature number SPVS002) contains the recommended operating conditions, electrical specifications, and timing characteristics of the TMS34010.

- The ***TMS34010 C Compiler User's Guide*** (literature number SPVU005) tells you how to use the TMS34010 C compiler. This C compiler accepts standard Kernighan and Ritchie C source code and produces TMS34010 assembly language source code. We suggest that you use *The C Programming Language* (written by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall) as a companion to the *TMS34010 C Compiler User's Guide.*

- The ***TMS34010 Software Development Board User's Guide*** (literature number SPVU002) describes using the TMS34010 software development board (a high-performance, PC-based graphics card) for testing and developing TMS34010-based graphics systems.

- The ***TMS34010 Software Development Board Schematics*** (literature number SPVU003) is a companion to the *TMS34010 Software Development Board User's Guide.*

- The ***TMS34010 Font Library User's Guide*** (literature number SPVU007) describes a set of fonts that are available for use in a TMS34010-based graphics system.

1.5 Style and Symbol Conventions

- In this document, program listings, program examples, interactive displays, filenames, and symbol names are shown in a `special font`. Examples use a **`bold version of the special font`** for emphasis. Here is a sample program listing:

```
0011  00000210    0001      .field    1, 2
0012  00000212    0003      .field    3, 4
0013  00000215    0006      .field    6, 3
0014  00000220              .even
```

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

  CVXYL *Rs, Rd*

  CVXYL is the instruction. This instructions has two parameters, indicated by *Rs* and *Rd Rs* and *Rd* are abbreviations for *source register* and *destination register*; when you use CVXYL, these parameters must be real register names (such as A0, B1, etc.).

- Square brackets ([and]) indicate an optional parameter. Here's an example of a directive that has an optional parameter:

  .field *value [, size in bits]*

  The **.field** directive has two parameters. The first parameter, *value*, is required. The second parameter, *size in bits*, is optional. As this syntax shows, if you use the optional second parameter, you must precede it with a comma.

  Square brackets are also used as part of the pathname specification for VMS pathnames; in this case, the brackets are actually part of the pathname (they aren't optional).

- Some parameters must be enclosed in double quotes. For example, consider the .sect directive:

  .sect *"section name"*

  This directive has one parameter, *section name*. When you use .sect, this parameter must be an actual section name, and it must be enclosed in double quotes.

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

  MM *address* { *16-bit-value* | *32-bit-value* | *assembler-statement* }

  The **MM** command has two parameters. The first parameter must be an *address*; the second parameter can be a *16-bit value*, a *32-bit value*, or an *assembler statement*.

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

  .byte $value_1$ *[, ... ,* $value_n$*]*

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

Section 2

Software Installation

This section contains step-by-step instructions for installing and executing the assembler, archiver, linker, object format converter, and simulator[2]. This software can be installed on four operating systems:

IBM and TI PCs

- PC-DOS[3] (IBM PC)
- MS-DOS[4] (TI PC)

Digital Equipment Corporation VAX-11[5]

- VMS operating system
- DEC Ultrix operating system
- Unix System V operating system

You will find installation instructions for these systems in the following sections:

Section 1.5 (page 1-7) describes style and symbol conventions that are used in this section.

---

2 The simulator is available in a PC version only.

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

4 MS is a trademark of Microsoft Corporation.

5 VAX-11 and VMS are trademarks of Digital Equipment Corporation.

2.1 Installation for IBM/TI PCs with PC/MS-DOS

The TMS34010 software package is shipped on several double-sided, double-density diskettes. Note that two versions of the simulator are shipped (one is for the IBM-PC and one is for the TI-PC). Use the simulator that is appropriate for your system; refer to the release notes for more information.

The tools execute in batch mode on PC-DOS (IBM PC) and MS-DOS (TI PC) systems. At least 512K bytes of memory space must be available in your system.

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

A: Floppy disk drive for hard-disk and single-drive 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 MS/PC-DOS prompts: `FORMAT ANOTHER (Y/N)?`, respond with `N`. Now copy the disks.

- On **hard-disk** or **single-drive** systems, enter:

```
DISKCOPY A:  A: CR
```

Follow the system prompts, removing and inserting the product and blank diskettes as directed. When MS/PC-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 B. Enter:

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

2) Create a directory to contain the TMS34010 software.

- On **hard-disk** or **single-drive** systems, enter:

```
MD E:\GSPTOOLS CR
```

- On **dual-drive** systems, enter:

```
MD B:\GSPTOOLS CR
```

3) Copy the TMS34010 tools onto the hard disk or the system disk. (Remember, you have two copies of the simulator disk – copy only one.)

- On **hard disk** or **single-drive** systems, enter:

  ```
  COPY A:\*.* E:\GSPTOOLS\*.* CR
  ```

- On **dual-drive** systems, enter:

  ```
  COPY A:\*.* B:\GSPTOOLS\*.* CR
  ```

2.2 Installation for VAX/VMS

The TMS34010 software tape was created with the VMS BACKUP utility at 1600 BPI. These tools were developed on version 4.4 of VMS. If you are using an earlier version of VMS, you must relink the object files; refer to the release notes for relinking instructions.

1) Mount the tape on your tape drive.

2) Execute the following commands. Note that you must create a destination directory for the tools; in this example, `DEST:directory` represents that directory. Replace `TAPE:` with the name of the tape drive you are using.

```
$ allocate              TAPE:
$ mount/for/den=1600    TAPE:
$ backup                TAPE:GSP.bck DEST:directory
$ dismount              TAPE:
$ dealloc               TAPE:
```

3) The product tape contains a file called `setup.com`. This file sets up VMS symbols that allow you to execute the tools in the same manner as other VMS commands. Execute the file as follows:

```
$ @setup                DEST:directory
```

This sets up symbols that you can use to call the various tools. As the file is executed, it will display the defined symbols on the screen.

2.3 Installation for VAX/ULTRIX and VAX/System V

This tape was made at 1600 BPI using the TAR utility. Follow these instructions to install the software:

1) Mount the tape on your tape drive.

2) Make sure that the directory that you'll store the tools in is the current directory.

3) Enter the TAR command for your system; for example,

```
TAR x
```

This copies the entire tape into the directory.

Section 3

Introduction to Common Object File Format

The assembler and linker create object files that are in a format called *common object file format*, or ***COFF***.

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

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

Appendix A details COFF object file structure; for example, it describes 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 the internal format of object files.

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 TMS34010 memory map. Each section of an object file is separate and distinct from the other sections. COFF object files always contain three default sections:

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

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

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

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

The assembler provides several directives that allow you to associate various portions of code and data with the appropriate sections. The assembler builds these sections during the assembly process, creating an object file that is organized similarly to the object file shown in Figure 3-1.

One of the linker's functions is to relocate 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 target memory. For example, you can define a section that contains an initialization routine, and then allocate the routine into a portion of the memory map that contains EPROM.

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

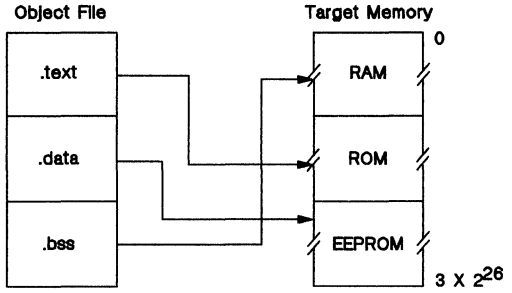


Figure 3-1. Partitioning Memory into Logical Blocks

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** and **.usect directives** reserve defined amounts of space in memory (usually RAM). This reserved space is used for storing variables.
- The **.text directive** identifies the source statements that follow it as executable code. The statements following a .text directive are assembled into the .text section.
- The **.data directive** identifies the source statements that follow it as initializable data. The statements following a .data directive are assembled into the .data section.
- The **.sect directive** defines named sections that are used like the .text and .data sections. The statements following a .sect directive are assembled into the named section.

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

> **Note:**
>
> If you don't use any of the sections directives, the assembler assembles everything into the .text section.

3.2.1 Uninitialized Sections

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

Uninitialized data areas are built by using the .bss and .usect assembler directives. The .bss directive reserves space in the .bss section. The .usect directive reserves space in a specific uninitialized named section. Each time you invoke the .bss directive, the assembler reserves more space in the .bss section. Each time you invoke the .usect directive, the assembler reserves more space in the specified named section.

The syntaxes for these directives are:

.bss *symbol, size in bits*

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

- The *symbol* points to the first bit reserved by this invocation of the .bss or .usect directive. The *symbol* corresponds to the name of the variable that you're reserving space for. It can be referenced by any other section and can also be declared as a global symbol (with the .global assembler directive).

- The *size* is an absolute expression. The .bss directive reserves *size* bits in the .bss section; the .usect directive reserves *size* bits in section *name*.

- The *section name* parameter tells the assembler which named section to reserve space in. (For more information about named sections, see Section 3.2.3.)

The .text, .data, and .sect directives tell the assembler to stop assembling into the current section and begin assembling into the indicated section. The .bss and .usect directives, however, **do not** end the current section and begin a new one; they simply "escape" from the current section temporarily. The .bss and .usect directives can appear anywhere in an initialized section without affecting the contents of the initialized 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 memory when the program is loaded. Each initialized section is separately relocatable and may reference symbols that are defined in other sections. The linker automatically resolves these section-relative references.

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

.text

.data

.sect *"section name"*

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

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

3.2.3 Named Sections

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

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

Two directives let you create named sections:

- The **.usect** directive creates sections that are used like the .bss section. These sections reserve space in RAM for variables.
- The **.sect** directive create sections that are used like the default .text and .data sections.

The syntaxes for these directives are:

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

.sect *"section name"*

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

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

3.2.4 Section Program Counters

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

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

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

3.2.5 An Example That Uses Sections Directives

Figure 3-2 shows how you can build COFF sections incrementally, using the sections directives to swap back and forth between the different sections. You can use sections directives:

- To begin assembling into a section for the first time, **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 listing file. By using a listing file, this example shows how the SPCs are modified during assembly. A line in a listing file has four fields:

1) The first field contains the source code line counter.
2) The second field contains the section program counter.
3) The third field contains the object code.
4) The fourth field contains the original source statement.

```
0001                    ***********************************************
0002                    **  Assemble an initialized table into .data  **
0003                    ***********************************************
0004 00000000                   .data
0005 00000000 00000040  pixvals: .long    64, 32, 16
     00000020 00000020
     00000040 00000010
0006             0005   pbuf_sz: .set     5    ; (produces no object code)
0007
0008                    ***********************************************
0009                    **  Reserve space in .bss for two variables   **
0010                    ***********************************************
0011 00000000                   .bss      var1, 32, 1
0012 00000020                   .bss      pix_buf, pbuf_sz * 16, 1
0013
0014                    ***********************************************
0015                    **                Still in .data             **
0016                    ***********************************************
0017 00000060 11111111  colors:  .long    011111111h, 022222222h, 033333333h
     00000080 22222222
     000000A0 33333333
0018                    ***********************************************
0019                    **    Assemble code into the .text section    **
0020                    ***********************************************
0021 00000000                   .text
0022 00000000     09E0  init_a:  MOVI     pix_buf, A0
     00000010 00000020+
0023 00000030     07A2           MOVE     @colors, A2,1
     00000040 00000060"
0024 00000060     2602           SLL      16, A2
0025 00000070     05A3           MOVE     @pixvals, A3
     00000080 00000000"
0026 000000A0     EE43           MOVY     A2, A3
0027 000000B0     09C4           MOVI     pbuf_sz, A4
     000000C0     0005
0028                    ***********************************************
0029                    **  Assemble another initialized table into   **
0030                    **        a named section called powers       **
0031                    ***********************************************
0032 00000000                   .sect     "powers"
0033 00000000 00000006  powers:  .long    6, 5, 4
     00000020 00000005
     00000040 00000004
0034                    ***********************************************
0035                    **  Define an uninitialized named section to  **
0036                    **      reserve more space for variables      **
0037                    ***********************************************
0038 00000000           var2:    .usect   "newvars", 1 * 16, 1
0039 00000010           inbuf:   .usect   "newvars", 8 * 16, 1
0040
0041                    ***********************************************
0042                    **       Assemble more code into .text        **
0043                    ***********************************************
0044 000000D0                   .text
0045 000000D0     9260  aloop:   MOVE     A3, *A0+, 1
0046 000000E0     3C44           DSJS     A4, aloop
```

Figure 3-2. Using Sections Directives

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

.text contains 240 bits of object code.

.data contains 192 bits of object code.

.bss reserves 112 bits in memory.

powers is a named section created with the .sect directive; it contains 96 bits of initialized data.

newvars is a named section created with the .usect directive; it reserves 144 bits in memory.

In Figure 3-3, note that the first column shows the source statements in Figure 3-2 that generate the object code in column 2.

Line Numbers	Object Code	Section
22	09E0	.text
22	00000020+	
23	07A2	
23	00000060"	
24	2602	
25	05A3	
25	00000000"	
26	EE43	
27	09C4	
27	0005	
45	9260	
46	3C44	
5	00000040	.data
5	00000020	
5	00000010	
17	11111111	
17	22222222	
17	33333333	
11, 12	No data- 112 bits reserved	.bss
33	00000006	powers (initialized named section)
33	00000005	
33	00000004	
38, 39	No data- 144 bits reserved	newvars (uninitialized named section)

Figure 3-3. Object Code Generated by Figure 3-2

3.3 How the Linker Handles Sections

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

The linker provides two directives that support these functions:

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

It is not always necessary to use linker directives. If you don't use them, the linker uses the default allocation algorithm described in Section 3.3.1. When you *do* use linker directives, you must specify them in a linker command file.

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

3.3.1 Default Allocation

You can link files without specifying a MEMORY or SECTIONS directive. The linker uses a default model to combine sections (if necessary) and allocate them into memory. When using the default model, the linker:

1) Assumes that memory begins at address 0, and assumes that 3×2^{26} words are available to allocate object code into.
2) Allocates .text into memory beginning at address 0.
3) Allocates .data into memory immediately following .text.
4) Allocates .bss into memory, immediately following .data.
5) Allocates any named sections, immediately following .bss. Named sections are allocated in the order that the linker encounters them in the input files.

Figure 3-4 shows how a *single* file would be allocated into memory using default allocation. Note that the linker does not actually place object code into memory; it assigns addresses to sections to that a *loader* can place the code into memory.

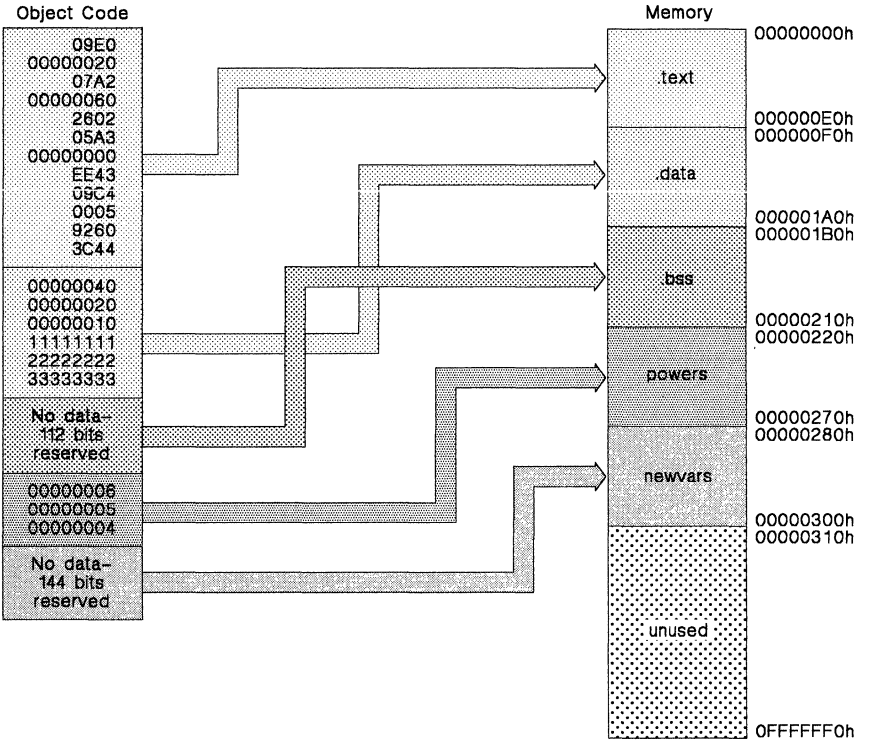


Figure 3-4. Default Allocation of the Object Code from Figure 3-2

As Figure 3-4 shows, the linker:

1) Allocates the .text section, beginning at address 0. The .text section contains 240 bits of object code.

2) Allocates the .data section next, beginning at address 180h. The .data section contains 192 bits of data.

3) Allocates the .bss section, beginning at address 380h. The .bss section reserves 112 bits in memory.

4) Allocates the initialized named section `powers` at address 390h. The `powers` section contains 96 bits of data.

5) Allocates the uninitialized named section `newvars` at address 4F0h. The `newvars` section reserves 144 bits in memory.

Note that the space between addresses 0300h–0FFFFFFF0h is not used.

Figure 3-5 shows a simple example of how *two* files would be linked together. When you link several files using the default algorithm, the linker combines all input sections that have the same name into one output section that has this same name. For example, the linker combines the .text sections from two input files to create one .text output section.

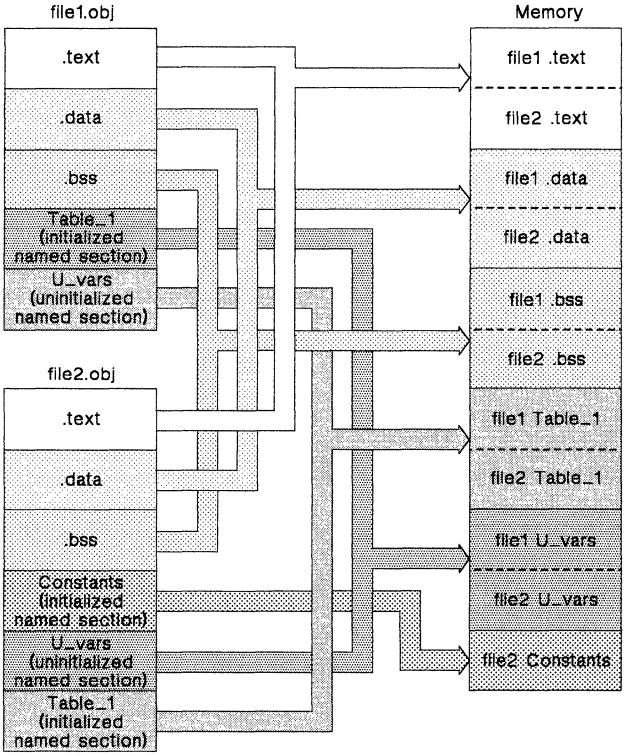


Figure 3-5. Combining Input Sections from Two Files (Default Allocation)

In Figure 3-5, `file1.obj` and `file2.obj` each contain the .text, .data, and .bss default sections, an initialized named section called `Table_1`, and an uninitialized named section called `U_vars`. `file2.obj` also contains an initialized named section called `Constants`. As Figure 3-5 shows, the linker:

1) Combines `file1` .text with `file2` .text to form one .text output section. The .text output section is allocated at address 0.

2) Combines `file1` .data with `file2` .data to form the .data output section. The .data output section is allocated following the .text output section.

3) Combines `file1` .bss with `file2` .bss to form the .bss output section. The .bss output section follows the .data section in memory.

4) Combines `file1 Table_1` with `file2 Table_1` to form the `Table_1` output section. (The `Table_1` section is the first named section that the linker encounters, so it is allocated before the other named sections.) The `Table_1` output section is allocated following the .bss output section.

5) Combines `file1 U_vars` with `file2 U_vars` to form the `U_vars` output section. The `U_vars` output section is allocated following the .bss output section.

6) Allocates the `Constants` section from `file2` after the `U_vars` section.

> **Note:**
>
> The maximum size of an output section is 0FFFFFFFh bits.

3.3.2 Placing Sections in the Memory Map

Figure 3-4 and Figure 3-5 illustrate the linker's default methods for combining sections and allocating them into memory. Sometimes you may not want to use the default setup. For example, you may not want to combine all of the .data sections into a single .data output section. Or, you might want to place a named section at address 0 instead of the .text section. Most memory maps are comprised of various types of memories (DRAM, VRAM, EPROM, etc.) in varying amounts; you may want to place a section in a particular type of memory.

The next three illustrations show another possible combination of the sections from Figure 3-4. Figure 3-6 contain MEMORY and SECTIONS definitions that define a memory map and allocate the sections from Figure 3-4 into the defined memory. Figure 3-7 shows how the ranges defined in Figure 3-6 fit into the memory map. Figure 3-8 shows how the sections from Figure 3-4 are allocated into the memory map.

```
MEMORY
{
    display :  origin = 0h          , length = 01FFFF0h
    code    :  origin = 0D0000000h , length = 03FFFF0h
    space   :  origin = 0FFE00000h , length = 01FFFF0h
}

SECTIONS
{
    .text  0D0000000h : { }
    powers ALIGN(32)  : { }  > code
    .data             : { }  > code
    newvars           : { }  > space
    .bss              : { }  > space
}
```

Figure 3-6. MEMORY and SECTIONS Directives for Figure 3-7 and Figure 3-8

In Figure 3-6,

- The MEMORY directive defines three memory ranges: `display`, `code`, and `space`.

  The *origin* for each of these ranges identifies the memory range's starting address in memory. The *length* specifies the length of the range.

  For example, memory range `space` has a starting address 0FFE00000h and a length of 01FFF0h; it defines the addresses 0FFE00000h through 0FFFFFFF0h in memory.

  Note that this MEMORY definition *does not define* the addresses between 01FFFF0h and 0CFFFFFFFh. This range is **unconfigured**. As far as the linker is concerned, this area does not exist, and no code or data can be loaded into it. Whenever you use the MEMORY directive, only the memory ranges that the directive defines can contain code or data.

- The SECTIONS directive defines the order in which the sections are allocated into memory. The .text section must begin at address 0D0000000h. The `powers` section is allocated into the named memory range `code` Since `code` starts at 0D0000000h and .text starts at address 0D0000000h, `powers` will follow .text; it must be aligned on the next available address that is a multiple of 32 bits. The .data section is allocated into memory range `code` following `powers`. The `newvars` section is allocated into the first available address in the named memory range `space`; the .bss section follows `newvars`.

Figure 3-7 shows how the ranges defined by the MEMORY directive in Figure 3-6 fit into the memory map.

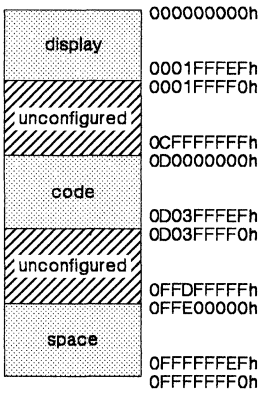


Figure 3-7. Memory Map Defined in Figure 3-6

Figure 3-8 shows how the sections from Figure 3-4 are allocated into the memory ranges defined in Figure 3-6. Note that some of the memory is configured but unused. For example, the `newvars` and .bss sections are allocated into the `space` area. The length of the `space` range is 01FFFF0h words; however, the combined lengths of the `newvars` and .bss sections is only 100h words. Thus, locations 0FFE00100h–0FFFFFFF0h are unused.

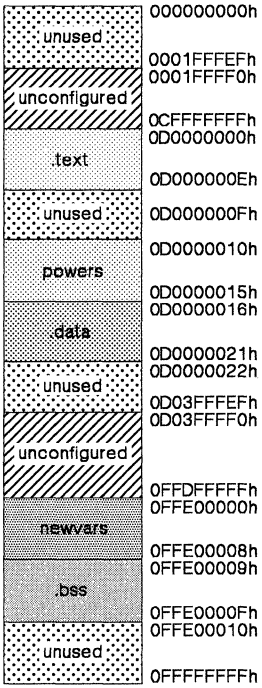


Figure 3-8. Placing the Code from Figure 3-4 into the Memory Map Defined by Figure 3-6

3.4 Relocation

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

- Allocating sections into the memory map so that they begin at the appropriate address,
- Adjusting symbol values to correspond to the new section addresses, **and**
- Adjusting references to relocated symbols to reflect the adjusted symbol values.

The linker uses *relocation entries* to adjust references to symbol values. The assembler creates a relocation entry each time a relocatable symbol is referenced. The linker then uses these entries to "patch" the references after the symbols are relocated. Figure 3-9 contains a code segment that generates relocation entries.

```
0001                                .ref   X
0002 00000000                       .text
0003 00000000     C080              JAUC   X       ; Creates a reloc. entry
     00000010 00000000!
0004 00000020     0300   Y:         NOP
0005 00000030     05A0              MOVE   @Y, A0 ; Creates a reloc. entry
     00000040 00000020'
```

Figure 3-9. An Example of Code that Generates Relocation Entries

In Figure 3-9, both symbols X and Y are relocatable. X is defined in some other module; Y is defined in the .text section of this module. When assembled, X has a value of 0 (the assembler assumes all undefined external symbols have values of 0) and Y has a value of 20h (relative to address 0 in the .text section). The assembler generates two relocation entries, one for X and one for Y. The reference to X is an external reference (indicated by the ! character in the listing). The reference to Y is to a relocatable symbol defined internally in the .text section (indicated by the ' character in the listing). After linking, suppose that X is relocated to address 100h. Suppose also that the .text section is relocated to begin at address 2000h; Y now has a relocated value of 2020h. The linker uses the two relocation entries to patch the two references in the object code:

```
    C080    JAUC X              becomes          C080
00000000                                     00000100
    05A0    MOVE @Y, A0         becomes          05A0
00000020                                     00002020
```

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

3.5 Loading a Program

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

In order to run a program, the data in the executable object module must be transferred, or **loaded**, into target system memory.

Several methods can be used for loading a program, depending on the execution environment. Some of the more common situations are listed below.

- Several of the TMS34010 debugging tools (such as the XDS emulator) have built-in loaders. Each of these tools has a LOAD command that invokes a COFF loader; the loader reads the executable file and copies the program into target memory.

- The TMS34010 software development board includes a stand-alone loader that allows you to load code into a target system, begin program execution, and then return to the operating system. The source for this loader is included with the SDB.

- If you are using a ROM- or EPROM-based system, you can use the object format converter (which is shipped as part of the assembly language package) to convert the executable COFF object module into one of several ASCII object file formats. You can then use the converted ASCII file with an EPROM programmer to burn the program into an EPROM.

- Some TMS34010 programs are loaded under the control of an operating system or monitor software running directly on the target system. In this type of application, the target system usually has an interface to the file system on which the executable module is stored. You must write a custom loader for this type of system. The loader must comprehend the file system (in order to access the file) as well as the memory organization of the target system (to load the program into memory).

3.6 Symbols in a COFF File

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

3.6.1 External Symbols

External symbols are symbols that are defined in one module and referenced in another module. You can use the .global directive to identify symbols as external. In a source module, an external symbol can be either:

- **Def**ined in the current module and used in another module, **or**
- Defined in another module and **ref**erenced in the current module.

The following code segment illustrates these definitions.

```
x:    .set     056h       ; Define x
      MOVE     @y, A1     ; Reference y
      .global  x          ; DEF of x
      .global  y          ; REF of y
```

The .global definition of x says that it is an external symbol defined in this module, and that other modules can reference x. The .global definition of y says that it is an undefined symbol that is defined in some other module.

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

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

3.6.2 The Symbol Table

The assembler always generates an entry in the symbol table when it encounters an external symbol (both definitions and references). The assembler also creates special symbols that point to the beginning of each section; the linker uses these symbols to relocate references to other symbols in a section.

The assembler does not usually create symbol table entries for any other type of symbol because the linker does not use them. For example, labels are not included in the symbol table unless they are declared with .global. For symbolic debugging purposes, it is sometimes useful to have entries in the symbol table for each symbol in a program. To accomplish this, invoke the assembler with the -s option.

Section 4

Assembler Description

The assembler translates assembly language source files into machine language object files. These files are in common object file format (COFF), discussed in Section 3. Source files can contain these assembly language elements:

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

The assembler:

- Is a two-pass assembler, with an intermediate pass for optimizing jump instructions.
- Processes the source statements in a text file to produce a relocatable object file.
- Produces a source listing (if requested) and provides you with control over the format of this listing.
- Appends a cross-reference listing to the source listing (if requested).
- Allows you to segment your code into sections.
- Maintains an **SPC** (section program counter) for each section of object code.
- Defines and references global symbols.
- Assembles conditional blocks.

Topics in this section include:

4.1 Assembler Development Flow

Figure 4-1 illustrates the assembler's role in the assembly language development flow. The assembler accepts assembly language source files as input; it also accepts assembly language files created by the C compiler.

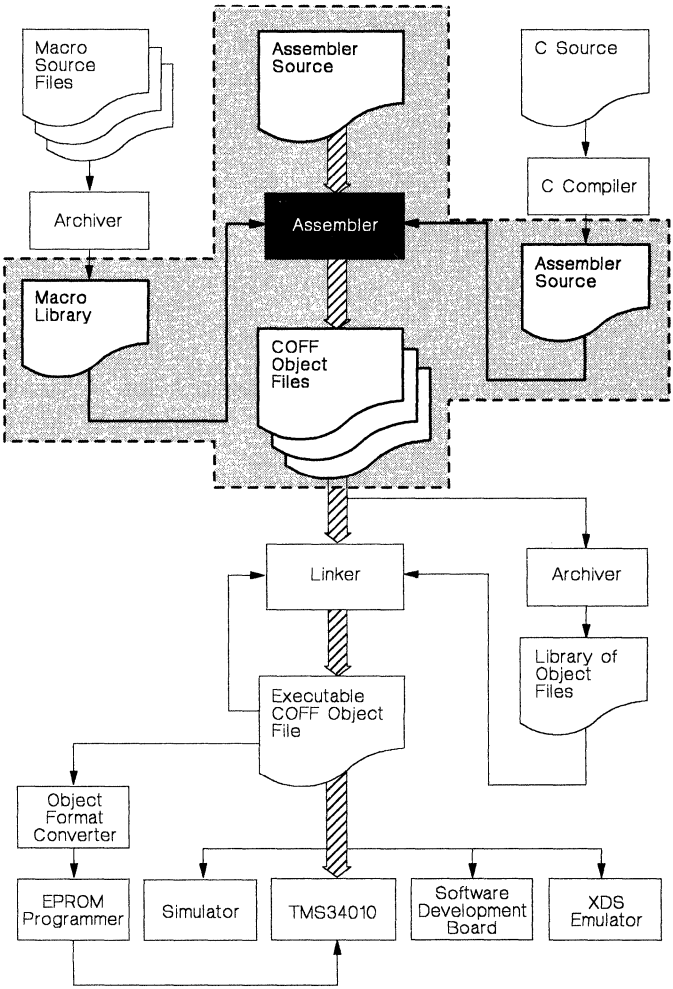


Figure 4-1. Assembler Development Flow

4.2 Invoking the Assembler

To invoke the assembler, enter:

gspa *input file [object file [listing file]] [-options]*

gspa is the command that invokes the assembler.

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

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

option identifies the assembler options that you want to use. *Case is insignificant for assembler options.* Options can appear anywhere on the command line; precede each option with a hyphen (-). You can string the options together; for example, **-lc** is equivalent to **-l -c**. Valid assembler options include:

- **-l** (lowercase "L") produces a listing file.
- **-i** specifies a directory name where the assembler can find files named by the .copy, .include, or .mlib directives. The format of the -i option is **-i***pathname*. You can specify up to 10 directories in this manner; each pathname must be preceded by the -i option.
- **-x** produces a cross-reference table and appends it to the end of the listing file. If you did not request a listing file, the assembler creates one anyway, but the listing will only contain the cross-reference table.
- **-c** makes case insignificant in the source file. For example, the symbols ABC and abc will be equivalent. *If you do not use this option, case is significant.*
- **-s** Put all defined symbols in the object file's symbol table.
- **-q** (quiet) suppresses the banner and all progress information.
- **-b** makes blanks significant. This option provides compatibility with older code in which a blank terminated the operand field, and thus began the comment field.
- **-h** allows hexadecimal constants to use the format >*nnn*. This option provides compatibility with older code that uses this format. Note that this format limits the number of legal operators for expressions.

4.3 Specifying Alternate Directories for Assembler Input

The .copy and .include directives tell the assembler to read source statements from another file and the .mlib directive names a library that contains macro definitions. Section 5, Assembler Directives, provides examples of the .copy, .include, and .mlib directives. The syntax for these directives is:

.copy *"filename"*
.include *"filename"*
.mlib *"filename"*

The *filename* names a copy/include file that the assembler reads statements from or a macro library that contains macro definitions. The *filename* may be a complete pathname or a filename with no path information. If you provide a pathname, the assembler uses that path and *does not look* for the file in any other directories. If you do not provide path information, the assembler searches for the file in:

1) The directory that contains the current source file. (The current source file refers to the file that is begin assembled when the .copy, .include, or .mlib directive is encountered.)
2) Any directories named with the -i assembler option.
3) Any directories set with the environment variable A_DIR.

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

4.3.1 -i Assembler Option

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

gspa -i*pathname* *source-file*

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

Assume that a file called `source.asm` is in the current directory; `source.asm` contains the following directive statement:

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

The complete path/filename for `copy.asm` is:

- `c:\gsp\files\copy.asm` (DOS),
- `[gsp.files]copy.asm` (VMS), **or**
- `/gsp/files/copy.asm` (UNIX).

This is how you invoke the assembler:

DOS: `gspa -ic:\gsp\files source.asm`

VMS: `gspa -i[gsp.files] source.asm`

UNIX: `gspa -i/gsp/files source.asm`

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

4.3.2 Environment Variable (A_DIR)

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

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

VMS: **assign A_DIR** *"pathname; another pathname ... "*

UNIX: **setenv A_DIR** *"pathname;another pathname ... "*

The *pathnames* are directories that contain copy/include files or macro libraries. You can separate the pathnames with a semicolon or with blanks. In assembly source, you can now use the .copy, .include, or .mlib directives without specifying any path information. If the assembler doesn't find the file in the parent directory or in directories named by -i, it searches the paths named by the environment variable.

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

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

Assume that the complete path and file information for these copy files is:

- `c:\340\files\copy1.asm` and `c:\gsys\copy2.asm` (DOS),
- `[340.files]copy1.asm` and `[gsys]copy2.asm` (VMS), **or**
- `/340/files/copy1.asm` and `/gsys/copy2.asm` (UNIX).

This is how you set the environment variables and invoke the assembler:

DOS:
```
set  A_DIR=c:\gsys;c:\exec\files;c:\test
gspa  -ic:\340\files  source.asm
```

VMS:
```
assign  A_DIR "[gsys];[exec.files];[test]"
gspa  -i/340/files  source.asm
```

UNIX:
```
setenv  A_DIR "/gsys;/exec/files;/test"
gspa  -i/340/files  source.asm
```

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

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

DOS: `set A_DIR=`

VMS: `deassign A_DIR`

UNIX: `setenv A_DIR ""`

4.4 Source Statement Format

TMS34010 assembly language source programs consist of source statements that may contain assembler directives, assembly language instructions, macro directives, and comments. Source statement lines may be as long as the source file format allows. The assembler reads up to 200 characters per line. If a statement contains more than 200 characters, the assembler truncates the line and issues a warning.

The next several lines show examples of source statements:

```
SYM      .set   0A5h        ; Symbol SYM = 0A5h
Begin:   ADDI   SYM+5,A1    ; Add (SYM+5) to the contents of A1
         MOVE   A1,A2       ; Move contents of A1 to A2
```

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

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

where:

- Statements must begin with a label, a blank, or a comment indicator.
- Labels are optional; if used, they must begin in column 1.
- One or more blanks must separate each field. (Note that tab characters are equivalent to blanks.)
- Comments are optional. Comments that begin in column 1 can begin with an asterisk or a semicolon (* or ;), but comments that begin in any other column **must** begin with a semicolon.

4.4.1 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 $). Labels are case sensitive, and the first character cannot be a number. A label can be followed by a colon (:); the colon is not treated as part of the label name. If you don't use a label, 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 .byte directive to initialize several bytes, a label would point to the first byte. In the following example, the label `Start` has the value 40h.

```
 .       .          .            .
 .       .          .            .
 .       .          .            .
0009 00000030           * Assume some other code has been assembled
0010 00000040      0A   Start:  .byte 0Ah,3,7
     00000048      03
     00000050      07
```

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   .set  $  ; ($ is the current value of the SPC)
```

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

```
0003 00000050              Here:
0004 00000050       03             .byte  3
```

Since the TMS34010 is a bit-addressable machine, some operations (such as .field) can increment the SPC in a way that may not align it to point to a word boundary. Other directives (such as .even) automatically realign the SPC. If a statement that contains only a label is followed by a statement that realigns the SPC, then the label will not point to the statement that follows it. In this example, a .field directive "misaligns" the SPC; the label does not point to the statement that contains the .even directive.

```
0001 00000000                      .space  16
0002 00000010         03           .field  3,2
0003 00000012              Label:                ; Label = 12h
0004 00000020                      .even         ; SPC is now 20h
```

4.4.2 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 ADD, FILL, MOVE)
- Assembler directive (such as .data, .align, .set)
- Macro directive (such as $MACRO, $LOOP, $NOLOOP)
- A macro invocation

4.4.3 Operand List

The operand field is a list of operands that follows the mnemonic field. An operand can be a constant (see Section 4.5), a symbol (see Section 4.7), or a combination of constants and symbols in an expression (see Section 4.8). You must separate operands with commas.

4.4.4 Comment Field

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

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

4.5 Constants

The assembler supports seven types of constants:

- Binary integer constants,
- Octal integer constants,
- Decimal integer constants,
- Hexadecimal integer constants,
- XY constants,
- Character constants, **and**
- Assembly-time constants.

The assembler maintains each constant internally as a 32-bit quantity. The assembler determines the length of variable-length instructions (such as ADDI) so they can be implemented efficiently. The assembler will use a 16-bit or 32-bit operand, depending on the magnitude of the operand.

Note that constants **are not sign extended**, they are right justified. For example, the constant 0FFFFH is equal to $0000FFFF_{16}$ or $65{,}535_{10}$; it **does not** equal -1.

4.5.1 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 right-justifies the bits. Examples of valid binary constants include:

00000000B Constant equal to 0

0100000b Constant equal to 32_{10} (20_{16})

01b Constant equal to 1

11111000B Constant equal to 248_{10} ($0F8_{16}$)

4.5.2 Octal Integers

An octal integer constant is a string of up to 12 octal digits (0 through 7) followed by the suffix **Q**(or **q**). Examples of valid octal constants include:

10Q Constant equal to 8_{10} (8_{16})

100000Q Constant equal to $32{,}768_{10}$ (8000_{16})

226Q Constant equal to 150_{10} (96_{16})

4.5.3 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} ($3E8_{16}$)

-32768 Constant equal to $-32{,}768_{10}$ (8000_{16})

25 Constant equal to 25_{10} (19_{16})

4.5.4 Hexadecimal Integers

A hexadecimal integer constant is a string of up to 8 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 8 hexadecimal digits are specified, the assembler right-justifies the bits. Examples of valid hexadecimal constants include:

78h Constant equal to 120_{10} (0078_{16})

0Fh Constant equal to 15_{10} ($000F_{16}$)

37ACH Constant equal to $14{,}252_{10}$ ($37AC_{16}$)

4.5.5 XY Constants

An XY constant is a constant in the form [Y-expression, X-expression]. The X-expression and Y-expression can be any legal, well-defined 16-bit expression in the range [-32767, -32767] to [65535, 65535]. The Y part and the X part are concatenated to form a 32-bit field; the Y part occupies the 16 MSBs of the field and the X part occupies the 16 LSBs. Examples of valid XY constants include:

[1+2, 4] Constant equal to 00030004_{16}

[-1, 0] Constant equal to $FFFF0000_{16}$

[10q, 101b] Constant equal to 00080005_{16}

['ab', 'cd'] Constant equal to 62616463_{16}

[0Fh, 0Ch] Constant equal to $000F000C_{16}$

4.5.6 Character Constants

A character constant is a string of 1 to 4 characters enclosed in *single* quotes. The characters are represented internally as 8-bit ASCII characters. Two consecutive single quotes are required to represent each single quote within a character constant. A character constant consisting only of two single quotes (no character) is valid and is assigned the value 0. If less than 4 characters are specified, the assembler right-justifies the bits. Examples of valid character constants include:

'ab' Represented internally as 00006261_{16}

'C' Represented internally as 00000043_{16}

'''D' Represented internally as 00004427_{16}

'abcd' Represented internally as 64636261_{16}

Note the difference between character *constants* and character *strings* (Section 4.6 discusses character strings). A character constant represents a single integer value; a string is a list of characters.

4.5.7 Assembly-Time Constants

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

```
val     .set    3
        MOVI    val, A0
```

If you assign an integer constant to a symbol, then the symbol can only be used as an integer constant.

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

```
COLOR0  .set    B8
        MOVI    11111111h, COLOR0
```

4.6 Character Strings

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

Character strings are used for:

- Filenames (as in `.copy "filename"`)
- Section names (as in `.sect "section name"`)
- Data initialization directives (as in `.byte "charstring"`)

4.7 Symbols

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

Symbols that are used as **labels** become symbolic addresses that are associated with locations in the program. Labels **must** be unique; 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, .set, or .bss directive.

The assembler has several predefined symbols, including:

- **$**, the dollar sign character, which represents the current value of the section program counter.
- **SP** or **sp**, the stack pointer symbol, which refers to the 16th register in either register file A or B.
- Register symbols, which have the form **A***n*, **a***n*, **B***n*, or **b***n*, where *n* is 0–14.

4.8 Expressions

An expression is a constant, a symbol, or a series of constants and symbols separated by arithmetic operators. The range of valid expression values is -2,147,483,647 to 4,294,967,295.

Three main factors influence the order of expression evaluation:

- **Parentheses**: Expressions that are enclosed in parentheses are always evaluated first.

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

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

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

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

- **Left-to-right evaluation**: When parentheses 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

4.8.1 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
~	1s complement	^	Bitwise exclusive-OR
		|	Bitwise OR
		&	Bitwise AND
Group 2 Left-to-Right Evaluation		**Group 4 (Relational Operators) Left-to-Right Evaluation**	
*	Multiplication	<	Less than
/	Division	>	Greater than
%	Modulo	<=	Less than or equal to
<<	Left shift	>=	Greater than or equal to
>>	Right shift	=	(==) Equal to
		!=	Not equal to

Note: Operators in parentheses indicate an alternate form.

4.8.2 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 in multiplication.

4.8.3 Well-Defined Expressions

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

An example of a well-defined expression is:

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

4.8.4 Conditional Expressions

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

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

These operations have the lowest precedence; however, each has the same precedence within the group, so they are evaluated left to right. Conditional expressions evaluate to 1 if true and 0 if false.

4.8.5 Relocatable Symbols and Legal Expressions

Table 4-2 summarizes valid operations on absolute, relocatable, and external symbols. An expression cannot multiply or divide by a relocatable or external symbol. An expression cannot contain unresolved symbols that are relocatable with respect to different sections.

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.

Here are some examples of expressions that use relocatable and absolute symbols. These examples use four symbols that are defined as follows:

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

- **Example 1:**

  The statements in this example use an absolute symbol, LAB1. The first statement puts the value 51 into register A0. The second statement puts the value 27 into register A0.

```
MOVI   LAB1 + ((4+3) * 7), A0     ; A0 = 51
MOVI   LAB1 + 4 + 3 * 7, A0       ; A0 = 27
```

- **Example 2:**

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

  relocatable symbol ± *absolute symbol*

  or

  absolute value

  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.

```
MOVI   extern_1 - 10, A0       ; Legal
MOVI   10-extern_1, A0         ; Can't negate reloc. symbol
MOVI   -(intern_1), A0         ; Can't negate reloc. symbol
MOVI   extern_1/10, A0         ; / isn't an additive operator
MOVI   intern_1 + extern_1, A0 ; Multiple relocatables
```

- **Example 3:**

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

```
MOVI   intern_1 - intern_2 + extern_1, A0  ; Legal
MOVI   intern_1 + intern_2 + extern_1, A0  ; 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.

```
MOVI   intern_1 + extern_1 - intern_2, A0  ; Illegal
```

4.9 Source Listings

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

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

Each line in the source file produces a line in the listing file that contains a source statement number, an SPC value, the object code assembled, and the source statement. A source statement may produce more than one word of object code. The assembler prints the SPC value and object code on a separate line for each additional word. Each additional line is printed immediately following the source statement line.

```
1    2         3          4
0015           00000000'  FloatA  .set    $           ; Program entry point
0016 00000000      01E0           PUSHST              ; Push status register
0017 00000010      098F           MMTM    SP,A0,A14   ; Save registers used
```

Field 1 *Source Statement Number.* The source statement number is a 4-digit decimal number. The assembler numbers source lines as it encounters them in the source file; some statements increment the line counter but are not listed (for example, .title statements and statements following a .nolist are not listed). The difference between two consecutive source line numbers indicates the number of source lines entered that are not listed. Source lines generated by a macro call, a .copy directive, or an .include directive are renumbered starting at 0001. The original sequence continues after the copying or macro expansion is complete. The assembler precedes the line numbers of copied files with a letter code to identify the level of copying. An **A** indicates the first level, **B** indicates a second level, etc.

Field 2 *Section Program Counter.* This field contains the section program counter, or **SPC**, value (hexadecimal). Each section (.text, .data, .bss, and named sections) maintains a separate SPC. Some directives do not affect the SPC; they leave this field blank.

Field 3 *Object Code.* This field contains the hexadecimal representation of the object code. All machine instructions and directives use this field to list object code. This field also indicates the relocation type by appending one of the following characters to the end of the field:

- ! Undefined external reference
- ' .text relocatable
- " .data relocatable
- + .data relocatable
- - .bss, .usect relocatable

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

```
GSP COFF Assembler,  Version x.xx, 86.200            Mon Mar 16 15:15:28 1987
(c) Copyright 1985, 1986, Texas Instruments Inc.
                                                                    PAGE    1

0001                    ***************************************************
0002                    * This function takes the ABS of a single-precision,
0003                    * IEEE-format floating-point number. The value to be
0004                    * ABS is popped off the stack as described in the
0005                    * TMS34010 C Compiler User's Guide. The returned ABS
0006                    * number is pushed onto the stack as also specified
0007                    * in the guide.
0008                    ***************************************************
0009                    *
0010                            .file   "FloatA"
0011                    *
0012                    * Define as a subroutine
0013                    *
0014                            .global FloatA
0015           00000000' FloatA   .set    $                ; Program entry point
0016 00000000     01E0            PUSHST                   ; Push status register
0017 00000010     098F            MMTM    SP,A0,A14        ; Save registers used
     00000020     8002
0018 00000030     0541            SETF    1,0,0
0019 00000040     5600            XOR     A0,A0            ; Clear the sign bit
0020 00000050     B00E            MOVE    A0,*A14(-1),0    ; Put result on stack
     00000060     FFFF
0021 00000070     09AF            MMFM    SP,A0,A14        ; Restore registers
     00000080     4001
0022 00000090     01C0            POPST                    ; Restore status
0023 000000A0     0960            RETS
0024                              .end

No Errors,  No Warnings
```

Figure 4-2. Sample Assembler Listing

4.10 Cross-Reference Listings

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

```
GSP COFF Assembler,  Version x.xx, 86.200            Mon Mar 16 15:15:28 1987
(c) Copyright 1985, 1986, Texas Instruments Inc.
                                                                  PAGE    2


LABEL                           VALUE       DEFN    REF

INT0                            00000002+     15      2
INT1                            00000004+     16      2
INT2                            00000006+     17      2
ISR0                            REF                       4     15
ISR1                            REF                       4     16
ISR2                            REF                       4     17
RESET                           00000000+     13      2
RINT                            00000002+     26      3
TINT                            00000000+     25      3
VECS                            00000006+     28      3
XINT                            00000004+     27
```

Figure 4-3. Cross-Reference Listing Format

- The **label** column contains each symbol that was defined or referenced during the assembly.
- The **value** column contains an 8-digit hexadecimal number which is the value assigned to the symbol *or* a name that describes the symbol's attributes. A value may be followed by either a character that describes the symbol attributes. Table 4-3 lists these characters and names.
- The **definition** (DEFN) column contains the statement number in which the symbol is defined. This column is blank for undefined symbols.
- The **reference** (REF) column lists the line numbers of statements that reference the symbol. A blank in this column indicates that the symbol was never used.

Table 4-3. Symbol Attributes

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

Section 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
- Reserve space in memory for uninitialized variables
- Control the appearance of listings
- Initialize memory
- Assemble conditional blocks
- Define global variables
- Specify libraries that the assembler can obtain macros from
- Examine symbolic debugging information

This section is divided into two parts: the first part (Sections 5.1 through 5.7) describes the directives according to function, and the second part (Section 5.8) is an alphabetical reference. This section includes the following topics:

The TMS34010 C compiler uses several directives for symbolic debugging. Unlike other directives, symbolic debugging directives are not used in most assembly language programs. Appendix B discusses these directives; they are not discussed 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 directive syntax.

Table 5-1. Directives Summary

Directives that Affect the Current Section	
Mnemonic and Syntax	**Description**
.bss *symbol, size in bits [, word alignment flag]*	Reserve *size* bits in a named (uninitialized) section
.data	Assemble into the .data (initialized data) section
.sect *"section name"*	Assemble into a named (initialized) section
.text	Assemble into the .text (executable code) section
symbol **.usect** *"section name", size in bits [, word alignment flag]*	Assemble into the .text (executable code) section
Directives that Initialize Constants	
Mnemonic and Syntax	**Description**
.bes *size in bits*	Reserve *size* bits in the current section; note that a label points to the last bit of the reserved space
.byte $value_1$ *[, ...,* $value_n$*]*	Initialize one or more successive bytes in the current section
.double *floating-point value*	Initialize a 64-bit, double-precision, floating-point constant
.field *value [, size in bits]*	Initialize a variable-length field
.float *floating-point value*	Initialize a 32-bit, single-precision, floating-point constant
.int $value_1$ *[, ...,* $value_n$*]*	Initialize one or more 32-bit integers
.long $value_1$ *[, ...,* $value_n$*]*	Initialize one or more 32-bit integers
symbol **.set** *value*	Initialize an assembly-time constant
.space *size in bits*	Reserve *size* bits in current section; note that a label points to the first bit in the reserved space
.string *"*$string_1$*" [, ..., "*$string_n$*"]*	Initialize one or more text strings
.word $value_1$ *[, ...,* $value_n$*]*	Initialize one or more 16-bit integers
Directives that Align the Section Program Counter (SPC)	
Mnemonic and Syntax	**Description**
.align	Align the SPC on 32-word boundary
.even	Align the SPC on word (16-bit) boundary
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

Table 5-1. Directives Summary (Concluded)

Directives that Format the Output Listing (continued)	
Mnemonic and Syntax	**Description**
.nolist	Stop the source listing
.option { *B* | *D* | *F* | *L* | *M* | *T* | *X* }	Select output listing options
.page	Eject a page in the source listing
.title *"string"*	Print a title in the listing page heading
.width *page width*	Set the page width of the source listing
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**
.copy *["]filename["]*	Include source statements from another file
.def *symbol$_1$ [, ..., symbol$_n$]*	Identify one or more symbols that are defined in the current module and used in another module
.global *symbol$_1$ [, ..., symbol$_n$]*	Identify one or more global (external) symbols
.include *["]filename["]*	Include source statements from another file (similar to .copy, but the included statements are not listed)
.mlib *["]filename["]*	Define macro library
.ref *symbol$_1$ [, ..., symbol$_n$]*	Identify one or more symbols that are used in the current module but defined in another module
Miscellaneous Directives	
Mnemonic and Syntax	**Description**
.end	Program end
Symbolic Debugging Directives †	
Mnemonic and Syntax	**Description**
.block *beginning line number*	Begin a C block
.endblock *ending line number*	End a C block
.endfunc *ending line number*	End a function definition
.eos	End a structure, enumeration, or union definition
.etag *name, size*	Begin an enumeration definition
.file *"filename"*	Define a program identifier
.func *beginning line number*	Begin a function definition
.line *line number [, address]*	Specify the line number of a C source statement
.member *name, value [, type, storage class, size, tag, dims]*	Define a member of a structure, enumeration, or union
.stag *name, size*	Begin a structure definition
.sym *name, value [, type, storage class, size, tag, dims]*	Specify symbolic debug information for a global variable, local variable, or a function
.utag *name, size*	Begin a union definition

† Appendix B discusses symbolic debugging directives

5.2 Sections Directives

Section 3 discusses COFF sections in detail. Five directives associate the various portions of an assembly language program with the appropriate section:

- The **.bss** directive reserves space in the .bss section for variables.
- The **.usect** directive reserves space in an uninitialized named section. The .usect directive is similar to the .bss directive, but it allows you to reserve space separately from the .bss section.
- 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 creates an initialized named section and associates subsequent code or data with that section.

Figure 5-1 shows how you can use section directives to associate code and data with the proper sections. This is an output listing; column 1 shows line numbers, and column 2 shows the section program counter. Each section has its own program counter, or SPC. When code is first placed in a section, its SPC equals 0. When you resume assembling into a section, its SPC will resume counting as if there had been no intervening code.

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

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

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

```
0001                                    ********************************************
0002                                    *  Begin assembling into the .text section  *
0003                                    ********************************************
0004 00000000                                .text
0005 00000000      0001                      .word     1, 2
     00000010      0002
0006 00000020      0003                      .word     3, 4
     00000030      0004
0007
0008                                    ********************************************
0009                                    *  Begin assembling into the .data section  *
0010                                    ********************************************
0011 00000000                                .data
0012 00000000      0009                      .word     9, 10
     00000010      000A
0013 00000020      000B                      .word     11, 12
     00000030      000C
0014
0015                                    ********************************************
0016                                    *  Start assembling into a named, initial-  *
0017                                    *  ized section, var_defs                   *
0018                                    ********************************************
0019 00000000                                .sect     "var_defs"
0020 00000000      0011                      .word     17, 18
     00000010      0012
0021
0022                                    ********************************************
0023                                    *  Resume assembling into the .data section *
0024                                    ********************************************
0025 00000040                                .data
0026 00000040      000D                      .word     13, 14
     00000050      000E
0027
0028 00000000                                .bss      sym, 19    ; Reserve space
0029                                                               ; in .bss
0030 00000060      000F                      .word     15         ; Still in .data
0031
0032                                    ********************************************
0033                                    *  Resume assembling into the .text section *
0034                                    ********************************************
0035 00000040                                .text
0036 00000040      0005                      .word     5, 6
     00000050      0006
0037
0038 00000000                           usym .usect    "xy", 20   ; Reserve space
0039                                                               ; in section xy
0040 00000060      0007                      .word     7          ; Still in .text
```

Figure 5-1. Sections Directives Example

5.3 Directives that Initialize Constants

Several directives initialize constants:

- The **.set** 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 numeric constant (for example, in expressions).

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

```
0001                    0004  val   .set   4
0002 00000000             04        .byte  val, val*2, val+12
     00000008             08
     00000010             10
```

  Note that the .set directive produces no object code.

- The **.byte** directive places one or more 8-bit values into consecutive bytes in the current section. The first byte is always aligned on an 8-bit boundary.
- The **.word** directive places one or more 16-bit values into consecutive words in the current section. The first word is aligned on a 16-bit boundary.
- The **.int** and **.long** directives place one or more 32-bit values into consecutive locations in the current section. The first integer is aligned on a 16-bit boundary.
- The **.float** and **.double** directives initialize floating-point numbers. .float initializes a 32-bit, single-precision value, and .double initializes a 64-bit, double-precision value.
- The **.string** directive places 8-bit characters from one or more character strings into consecutive bytes in the current section. The characters are maintained as 8-bit ASCII codes.

  Figure 5-2 compares the .byte, .word, .int, .long, .float, .double, and .string directives; for this example, assume the following code has been assembled:

```
0003
0004 00000000       AB            .byte     0ABh, 0CDh
     00000008       CD
0005 00000010     CDEF            .word     0CDEFh
0006 00000020 12345678            .int      12345678h
0007 00000040 12345678            .long     12345678h
0008 00000060 E9C22CA9            .float    -1.0e25
0009 00000080 16140148            .double   -1.0e25
     000000A0 C5384595
0010 000000C0       47            .string   "GSP"
     000000C8       53
     000000D0       50
```

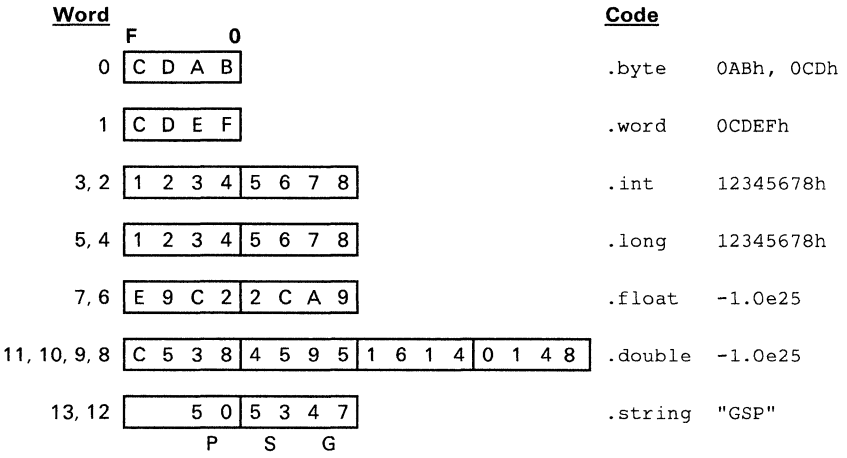


Figure 5-2. Examples of Initialization Directives

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

Figure 5-3 shows an example of the .space and .bes directives. Assume the following code has been assembled for this example:

```
0041 00000560      0100            .word     100h, 200h
     00000570      0200
0042 00000580            label1:   .space    38h
0043 000005FF            label2:   .bes      47h
```

`label1` points to the first bit reserved by .space. `label2` points to the last bit reserved by .bes.

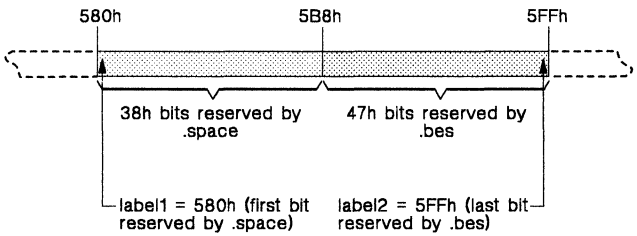


Figure 5-3. An Example of the .space and .bes Directive

- The **.field** directive places a single value into a specified number of bits in the current word. You can pack multiple fields into a single word, and fields can span words; the assembler does not align the SPC before or after it puts the value into memory.

  Figure 5-4 shows how fields are packed into a word. For the example, assume the following code has been assembled:

```
0004 00000000      08          .byte     8
0005 00000008      0003        .field    3, 2
0006 0000000A      0009        .field    9, 4
0007 0000000E      0006        .field    6, 7
0008 00000015      002C        .field    44, 9
```

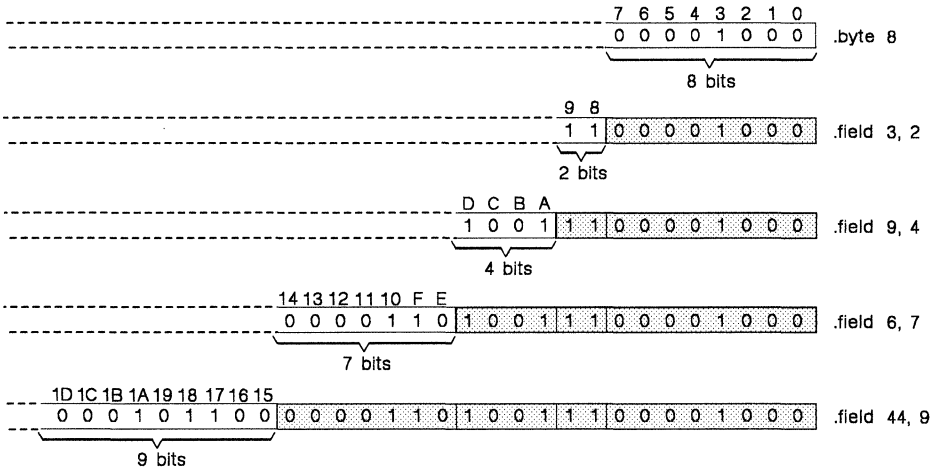


Figure 5-4. An Example of the .field Directive

5.4 Directives that Align the Section Program Counter

Several directives automatically align the SPC; for example, the .word directive will align the SPC if necessary so that the initialized word is aligned on a 16-bit boundary. The assembler also supports two directives that allow you to explicitly align the SPC on a word or cache boundary:

- The **.even** directive aligns the section program counter on a 16-bit boundary. After assembling a .byte, .field, .space, or .bes directive, the SPC may not be aligned on a 16-bit boundary. You can use the .even directive to force the assembler to align the SPC on the next word boundary; the assembler fills the space between the current SPC and the new SPC with 0s. If the SPC is already aligned on a word boundary, the .even directive does not affect the SPC.

  Figure 5-5 shows how the .even directive aligns the SPC. Assume the following code has been assembled:

```
0001                          * Part 1
0002 00000210      0001                .field    1, 2
0003 00000212      0003                .field    3, 4
0004 00000216      0006                .field    6, 3
0005 00000220                          .even
0006                          * Part 2
0007 00000220        08                .byte     8
0008 00000228                          .space    10h
0009 00000240                          .even
```

  In part 1, 9 bits of a word are filled by .field directives. The .even directive fills the remainder of the word with 0s and aligns the SPC on the next 16-bit boundary. In part 2, 8 bits of a word are filled by a .byte directive; the remaining 8 bits, and 8 bits of the next word, are filled by the .space directive. The .even directive fills the remainder of the second word with 0s and aligns the SPC on the next 16-bit boundary.

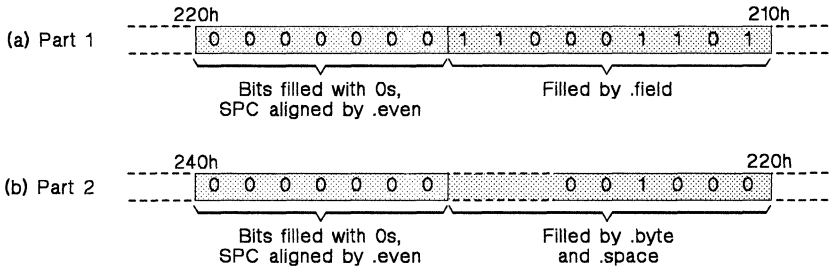


Figure 5-5. An Example of the .even Directive

- The **.align** directive aligns the SPC at the beginning of the next block of 32 16-bit instruction words. If the space between the current SPC and the new SPC is 3 words or less, the assembler fills this space with NOPs; otherwise, it inserts a jump instruction to the new SPC and fills the intervening space with NOPs.

Figure 5-6 shows an example that uses the .align directive. In this example, the SPC initially points to address 140h. After assembling the .align directive, the SPC points to the next 32-word boundary (address 200h). Assume the following code has been assembled:

```
0008 00000120        AA          .byte     0AAh
0009 00000130      0056          .word     56h, 23q
     00000140      0013
0010 00000200                    .align
0011 00000200      9801          MOVE      *A0+, *A1+
```

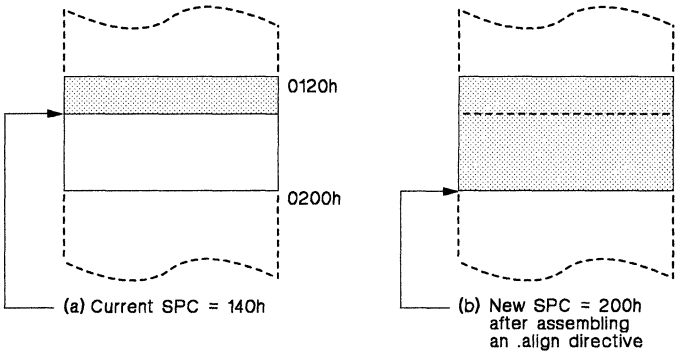


Figure 5-6. An Example of the .align Directive

The primary application for the .align directive is to align the SPC so that the code that follows .align starts on a cache boundary. The TMS34010 instruction cache is composed of 256 bytes of RAM, organized into four cache segments. Each cache segment is organized into eight blocks of four 16-bit instruction words, or 32 16-bit instruction words total. Figure 5-7 shows cache segment organization.

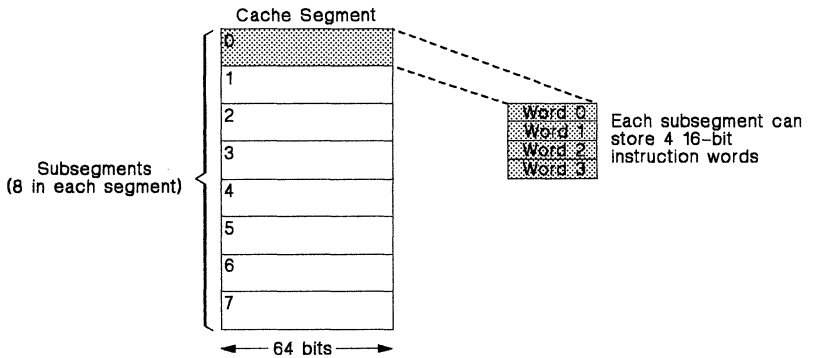


Figure 5-7. Cache Segment Organization

5.5 Directives that Format the Output Listing

Seven directives 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. By default, the assembler acts as if it has assembled a .list directive. 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. These directives can be nested.
- The **.mlist** and **mnolist** directives allow and inhibit macro expansion listings. These directives can be nested.
- The **.option** directive controls several features in the listing file. This directive has several operands:

  B Limits the listing of .byte directives to 1 line.
  D Limits the listing of .word directives to 1 line.
  F Resets the B, D, L, M, and T directives.
  L Limits the listing of .long and .int directives to 1 line.
  M Turns off macro expansions.
  T Limits the listing of .string directives to 1 line.
  X Produces a cross-reference listing of symbols. (You can also obtain a cross-reference listing by invoking the assembler with the -x option.)

- The **.page** directive causes a page eject in the output listing.
- The **.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 assembles the code that follows it (up to an .else or .endif).
  - If this expression evaluates to *false* (0), then the assembler assembles 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 assembles if the if-expression is false (0). This directive is optional in the conditional block; if an expression is 0 and there is no .else statement, then the assembler continues with the code that follows the .endif.
- The **.endif** directive terminates a conditional block.

The assembler supports several relational operators that are especially useful for conditional expressions; see Section 4.8.4 (page 4-14) for more information about relational operators.

Figure 5-8 shows an example of conditional assembly.

```
0004                    0001  sym1    .set     1
0005                    0002  sym2    .set     2
0006                    0003  sym3    .set     3
0007                    0004  sym4    .set     4
0008
0009                          If_1:   .if      sym1 < sym4
0010 00000000             01          .byte    sym1
0011                                  .else
0012                                  .byte    sym4
0013                                  .endif
0014                          If_2:   .if      sym1 + sym2 = sym4
0015                                  .byte    sym4
0016                                  .else
0017 00000008             01          .byte    sym4 - (sym1 + sym2)
0018                                  .endif
0019                          If_3:   .if      sym1 <> sym4 - sym2
0020 00000010             01          .byte    sym1
0021                                  .else
0022                                  .byte    sym4 - sym2
0023                                  .endif
```

Figure 5-8. An Example of Conditional Assembly

5.7 Directives that Reference Other Files

These directives supply information for or about other files:

- The **.copy** and **.include** directives tell the assembler to read source statements from another file. When the assembler is done reading the source statements in the copy/include file, it resumes reading source statements from the current file. The statements read from a copied file are printed in the listing file; the statements read from an included file are not printed in the listing file.

- The **.global** directive declares a symbol to be external so that it is available to other modules at link time. The .global directive does double duty, acting as a .def for defined symbols and as a .ref for undefined symbols. Note that the linker will resolve an undefined global symbol only if it is used in the program.

- The **.def** directive identifies a symbol that is defined in the current module and can be used by other modules. The assembler puts the symbol in the symbol table.

- The **.ref** directive identifies a symbol that is used in the current module and defined in another module. The assembler marks the symbol as an undefined external symbol and puts it in the object symbol table so the linker can resolve its definition.

- The **.mlib** directive supplies the assembler with the name of an archive library that contains macro definitions. When the assembler encounters a macro that is not defined in the current module, it searches for it in the 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 **.align**

Description The .align directive aligns the section program counter on the next 32-word boundary. This ensures that subsequent code can start on a cache boundary. If the space between the current SPC and the new (aligned) SPC is three words or less, the assembler fills this space with NOPs; otherwise, it inserts a jump instruction to the new SPC.

Using the .align directive has two effects:

- The assembler aligns the SPC on a 32-word (cache) boundary *within* the current section.
- The assembler sets a flag that forces the linker to align the section so that individual alignments remain intact when a section is loaded into memory.

Example This example aligns the SPC on the next 32-word boundary. This assures that the `clr_array` loop can start on a cache boundary.

```
0003                        ******************************************
0004                        *  Reserve space for a 100-element array.  *
0005                        *  The symbol array_start points to the    *
0006                        *  first address in the array.             *
0007                        ******************************************
0008                 0640   array_size:   .set    640h
0009 00000000               array_start:  .bss    temparray, array_size, 16
0010
0011                        ******************************************
0012                        *  Initialize the array with 0s; the loop  *
0013                        *  is aligned so it can start on a cache   *
0014                        *  boundary.                               *
0015                        ******************************************
0016 00000000       0550                    SETF    16, 0   ; FS0=0, FE0=0
0017 00000010       5621                    CLR     A1
0018 00000020       09E0                    MOVI    array_start, A0
     00000030 00000000+
0019 00000050       09C2                    MOVI    array_size, A2
     00000060       0640
0020 00000200                               .align          ; Align the loop
0021 00000200       9020   clr_array:       MOVE    A1,*A0+
0022 00000210       3C42                    DSJS    A2, clr_array
```

Syntax **.bss** *symbol, size in bits [, word alignment flag]*

Description The .bss directive reserves space in the .bss section for variables. This directive is usually used to allocate variables in RAM.

- The *symbol* is a required parameter. It defines a symbol that points to the first location reserved by the directive. The symbol name corresponds to the name of the variable that you're reserving space for.
- The *size* is a a required parameter; it must be an absolute expression. The assembler allocates *size* bits in the .bss section. There is no default size.
- The *word alignment flag* is an optional parameter. If you specify a nonzero value for the *alignment flag*, the assembler will align the reserved space on a 16-bit (word) boundary within .bss. If you specify a value of zero for the *alignment flag*, the assembler will not align the reserved space; this is also the default if no *alignment flag* is specified.

Other sections directives (.text, .data, and .sect) end the current section and begin assembling into another section. The .bss directive, however, does not affect the current section. The assembler assembles the .bss directive and then resumes assembling code into the current section. For more information about COFF sections, see Section 3.

Example This example uses the .bss directive to allocate space in .bss for two variables, `stack_size` and `C_array`.

```
0001                    ****************************************************
0002                    **            Begin assembling into .data         **
0003                    ****************************************************
0004 00000000                       .data
0005          00008000  stack_size:  .set     2048 * 16
0006
0007                    ****************************************************
0008                    **  Reserve 2k words in .bss for a software stack;  **
0009                    **  align this space on a word boundary. The symbol **
0010                    **  stack points to the stack's first address.      **
0011                    ****************************************************
0012 00000000                       .bss      stack, stack_size, 1
0013
0014                    ****************************************************
0015                    ** Still in .data. Set up register A14 as the soft- **
0016                    ** ware stack pointer; the symbol STK refers to A14.**
0017                    ****************************************************
0018              000E  STK:          .set      A14
0019 00000000     09EE              MOVI      stack, STK
     00000010 00000000+
0020
0021                    ****************************************************
0022                    **  Reserve space in .bss for a 100-element array.  **
0023                    **  Each element has a size of 2 bits; the array is **
0024                    **  not aligned on a word boundary.                 **
0025                    ****************************************************
0026 00008000                       .bss      C_array, 100 * 2, 0
0027
0028                    ****************************************************
0029                    **  Still in .data. Declare external .bss symbols.  **
0030                    ****************************************************
0031                                .global   stack, C_array
```

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

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

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

You can use as many values as fit on a single line. If you use a label, it points to the location at which the assembler places the first byte.

> **Note:**
>
> The first byte is always aligned on an 8-bit boundary.

Example This example places several 8-bit values into consecutive bytes in memory. The label `strx` has the value 20h, which is the location of the first initialized byte.

```
0001 00000000                    .space  020h
0002 00000020        32  strx:   .byte   32h, -1, 'A' + 1
     00000028        FF
     00000030        42
0003 00000038          61             .byte    "abc", 'D'*2
00000040       62
     00000048        63
     00000050        88
0004 00000058        61          .byte   'a'
0005 00000060        62          .byte   'b'
0006 00000068        63          .byte   'c'
```

Syntax

.copy *["]filename["]*

.include *["]filename["]*

(The quote marks surrounding the filename are optional.)

Description

The .copy and .include directives tell the assembler to read source statements from another file. The assembler:

1) Stops assembling statements in the main source file,

2) Assembles the statements in the copied/included file, **and**

3) Resumes assembling statements in the main source file, starting with the statement that follows the .copy or .include directive.

The *filename* is a required parameter that names a source file; the *filename* may be enclosed in double quotes. The *filename* must follow operating system conventions. You can specify a full pathname (for example, `.copy c:\gsp\file1.asm`). If you do not specify a full pathname, the assembler searches for the file in:

1) The directory that contains the current source file.

2) Any directories named with the -i assembler option.

3) Any directories specified by the environment variable A_DIR.

For more information about the -i option and the environment variable, see Section 4.3, Specifying Alternate Directories for Assembler Input, on page 4-4.

The statements that are assembled from a copy file are printed in the assembly listing. The statements that are assembled from an include file are *not* printed in the assembly listing, regardless of the number of .nolist/.list directives that are assembled.

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

Example 1 This example uses the .copy directive to assemble source statements from other files, then resumes assembling into the current file.

copy.asm (source file)

```
      .space  030h
      .copy   "byte2.asm"

** Back in original file
      .string "done"
```

byte2.asm (first copy file)

```
** In byte2.asm
      .byte   32, 64
      .copy   "word2.asm"
** Back in byte2.asm
      .byte   67h
```

word2.asm (second copy file)

```
** In word2.asm
      .word   0ABCDh, 56
```

Listing file:

```
 0001 00000000                      .space  030h
 0002                               .copy   "byte2.asm"
A0001                           ** In byte2.asm
A0002 00000030       20            .byte   32, 64
      00000038       40
A0003                              .copy   "word2.asm"
B0001                           ** In word2.asm
B0002 00000040     ABCD            .word   0ABCDh, 56
      00000050     0038
A0004                           ** Back in byte2.asm
A0005 00000060       67            .byte   67h
 0003
 0004                           ** Back in original file
 0005 00000068       64            .string "done"
      00000070       6F
      00000078       6E
      00000080       65
```

Example 2 This example is analogous to the first example; however, it uses the .include directive to assemble source statements from other files. Compare the listing files of these examples.

copy2.asm (source file)

```
  .space   030h
  .include "byte3.asm"

** Back in original file
  .string  "done"
```

byte3.asm (first include file)

```
** In byte3.asm
  .byte    32, 64
  .include "word3.asm"
** Back in byte3.asm
  .byte    67h
```

word3.asm (second include file)

```
** In word3.asm
  .word    0ABCDh, 56
```

Listing file:

```
0001 00000000                  .space   030h
0002                           .include "byte3.asm"
0003
0004                        ** Back in original file
0005 00000068       64         .string  "done"
     00000070       6F
     00000078       6E
     00000080       65
```

Syntax **.data**

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

Section 3 provides a detailed explanation about COFF sections.

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

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

```
0001                        ****************************************
0002                        **      Begin assembling into .data      **
0003                        ****************************************
0004 00000000                       .data                   ; ASCII table
0005 00000000     0000    asc_null: .word    00h
0006 00000010     0001    asc_soh:  .word    01h
0007 00000020     0002    asc_stx:  .word    02h
0008 00000030     0003    asc_etx:  .word    03h
0009 00000040     0004    asc_eot:  .word    04h
0010 00000050     0005    asc_enq:  .word    05h
0011 00000060     0006    asc_ack:  .word    06h
0012 00000070     0007    asc_bel:  .word    07h
0013 00000080     0008    asc_bs:   .word    08h
0014                        .global  asc_null, asc_soh, asc_stx
0015                        .global  asc_etx, asc_eot, asc_enq
0016                        .global  asc_ack, asc_bel, asc_bt
0017
0018                        ****************************************
0019                        **      Begin assembling into .text      **
0020                        ****************************************
0021 00000000                       .text
0022 00000000     0360              DINT
0023 00000010     0550              SETF     16, 0
0024
0025                        ****************************************
0026                        **     Resume assembling into .data      **
0027                        ****************************************
0028 00000090                       .data
0029 00000090     0041    asc_A:    .word    041h
0030 000000A0     0042    asc_B:    .word    042h
0031 000000B0     0043    asc_C:    .word    043h
0032 000000C0     0044    asc_D:    .word    044h
0033 000000D0     0045    asc_E:    .word    045h
0034 000000E0     0046    asc_F     .word    046h
0035                        .global  asc_A, asc_B, asc_C
0036                        .global  asc_D, asc_E, asc_F
```

Syntax **.double** *floating-point value*

Description The .double directive places a *64-bit, double-precision* representation of a floating-point constant into the current section. The C compiler uses the .double directive to initialize floating-point constants that can be manipulated with the C floating-point runtime support.

The *TMS34010 C Compiler User's Guide* describes the representation of floating-point numbers.

The floating-point value that is specified as an operand for the .double directive must be a floating-point constant. A floating-point constant is a string of decimal digits, followed by an optional decimal point, a fractional portion, and an exponent portion.

The syntax for a floating-point constant is:

[+|-] [nnn] [.nnn] [E|e [+|-] nnn]

nnn is a string of decimal digits.

Floating-point constants may be preceded by a + or a - sign; + is the default. Floating-point constants cannot be used with other arithmetic operators to form an expression; for example, 3.0 + 1 is illegal.

Example Here are some valid examples of the .double directive.

```
0010  000000E0  16140148        .double   -1.0e25
      00000100  C5384595
0011  00000120  00000000        .double   .5
      00000140  3FF80000
0012  00000160  00000000        .double   3
      00000180  401C0000
0013  000001A0  00000000        .double   -123
      000001C0  C06F6000
0014  000001E0  F80DC337        .double   3.14159
      00000200  401C90FC
0015  00000220  4BEC44DE        .double   -0.014E-14
      00000240  BCAE6959
0016  00000260  82000000        .double   36e10
      00000280  426A7A35
```

Syntax **.end**

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

Note that 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, the assembler would ignore them.

```
0001 00000000            Start:    .space  300
0002                0004 pix_size  .set    4
0003                0400 S_pitch   .set    256 * pix_size
0004                0400 D_pitch   .set    256 * pix_size
0005                0011 SPTCH     .set    B1
0006                0013 DPTCH     .set    B3
0007           C0000150  PSIZE     .set    0C0000150h
0008 00000130       1880           MOVK    pix_size, A0
0009 00000140       0580           MOVE    A0, @PSIZE
     00000150 C0000150
0010 00000170       09D1           MOVI    S_pitch, SPTCH
     00000180       0400
0011 00000190       09D3           MOVI    D_pitch, DPTCH
     000001A0       0400
0012                               .end
```

Syntax .even

Description The .even directive aligns the section program counter on the next word (16-bit) boundary. Since the TMS34010 is a bit-addressable processor, the SPC may not always be aligned on a word boundary. For example, some directives, such as .field, .byte, .bes, and .space, may initialize only part of the word. You can follow any of these directives with the .even directive; this forces the assembler to write out a partially filled word. If the SPC is already aligned on a word boundary, then the .even directive does not affect it.

Note that there is an implied .even at the end of each section; this ensures that each section ends on an even word boundary.

Example This example uses the .field, .byte, and .space directives to fill portions of words. Figure 5-9 (page 5-24) shows how .even aligns the SPC.

```
0001                            ********************************
0002                            *    Initialize a 6-bit field    *
0003                            ********************************
0004 00000000      0008                 .field   08h, 6
0005
0006                            ********************************
0007                            *    Initialize a 5-bit field    *
0008                            ********************************
0009 00000006      000F                 .field   0Fh, 5
0010
0011                            ********************************
0012                            * The remaining bits in the word *
0013                            * are filled with 0s and the SPC *
0014                            * is aligned on the next word    *
0015                            ********************************
0016 00000010                           .even
0017
0018                            ********************************
0019                            * Initialize a byte in a word, & *
0020                            * align the SPC at the next word *
0021                            ********************************
0022 00000010        02                 .byte    02h
0023 00000020                           .even
0024
0025                            ********************************
0026                            * Reserve 17 bits in memory, then*
0027                            * align the SPC at the next word *
0028                            * past the reserved space        *
0029                            ********************************
0030 00000020                           .space   17
0031 00000040                           .even
```

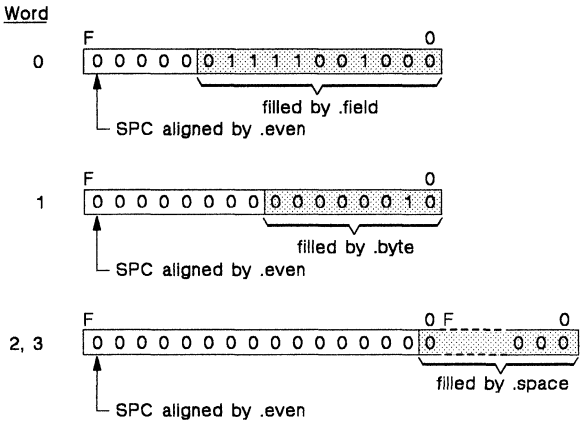


Figure 5-9. An Example of the .even Directive

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

Description The .field directive places one value into a specified number of bits in memory.

- The *value* is an expression that is evaluated and placed in the field. If the value is relocatable, *size* must be 32.
- The *size* is optional; it specifies a number from 1–32, which is the number of bits the field consists of. If you do not specify a size the assembler assumes that the size is 32 bits. If you specify a *value* that cannot fit in *size* bits, the assembler truncates the value and issues an error message.

> **Note:**
>
> The section program counter **is not aligned** before or after the value is placed in memory.

Example This example shows how multiple-bit fields are initialized. Figure 5-10 (page 5-26) shows how the fields are initialized.

```
0001                          ************************************
0002                          *     Initialize a 14-bit field    *
0003                          ************************************
0004 00000000      0ABC               .field   0ABCh, 14
0005
0006                          ************************************
0007                          *  Initialize a 5-bit field -- the  *
0008                          *  first two bits are in one word,  *
0009                          *  the next 3 bits are in another   *
0010                          ************************************
0011 0000000E      000A       I_F:    .field   0Ah, 5
0012
0013                          ************************************
0014                          *  Initialize a 4-bit field in the  *
0015                          *  same word                        *
0016                          ************************************
0017 00000013      000C               .field   0Ch, 4
0018
0019                          ************************************
0020                          *  Initialize a 3-bit field in the  *
0021                          *  same word                        *
0022                          ************************************
0023 00000017      0001       x:      .field   0lh, 3
0024
0025                          ************************************
0026                          *  Initialize a 32-bit relocatable  *
0027                          *  field in the next word           *
0028                          ************************************
0029 0000001A 00000017'               .field   x
```

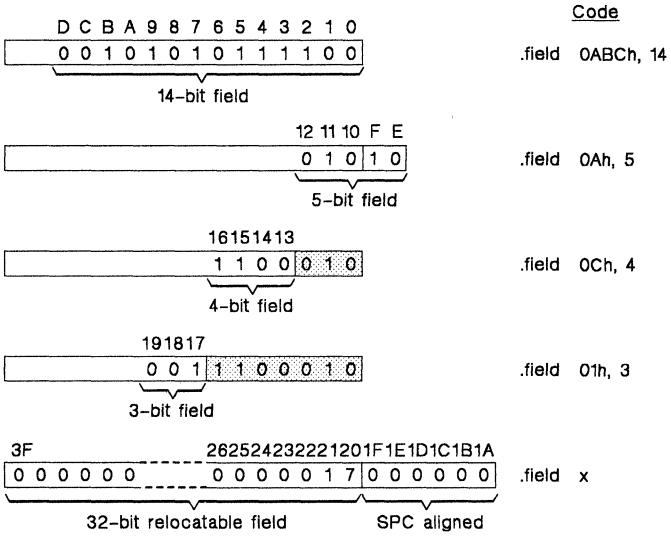


Figure 5-10. Examples of the .field Directive

Syntax **.float** *floating-point value*

Description The .float directive places a *32-bit, single-precision* representation of a floating-point constant into the current section. The C compiler uses the .float directive to initialize floating-point constants that can be manipulated with the C floating-point runtime support.

The *TMS34010 C Compiler User's Guide* describes the internal representation of floating-point numbers.

The floating-point value that is specified as an operand for the .float directive must be a floating-point constant. A floating-point constant is a string of decimal digits, followed by an optional decimal point, a fractional portion, and an exponent portion.

The syntax for a floating-point constant is:

[+|-] [nnn] [.nnn] [E|e [+|-] nnn]

nnn is a string of decimal digits.

Floating-point constants may be preceded by a + or a - sign; + is the default. Floating-point constants cannot be used with other arithmetic operators to form an expression; for example, 3.0 + 1 is illegal.

Example Here are some valid examples of the .float directive.

```
0001  00000000  E9C22CA9        .float  -1.0e25
0002  00000020  3FC00000        .float  .5
0003  00000040  40E00000        .float  3
0004  00000060  C37B0000        .float  -123
0005  00000080  40E487E8        .float  3.14159
0006  000000A0  A5734ACA        .float  -0.014E-14
0007  000000C0  5353D1AC        .float  36e10
```

Syntax

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

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

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

Description

The .global, .ref, and .def directives identify symbols that can be referenced externally.

- The **.def** directive identifies a symbol that is defined in the current module and can be accessed by other files. The assembler places this symbol in the symbol table.
- The **.ref** directive identifies a symbol that is defined in another module and used in the current module. The linker resolves this symbol's definition at link time.
- The **.global** directive acts as a .ref or a .def, as needed.

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

There are two main reasons for declaring symbols as global:

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

2) If the symbol *is defined in the current module*, then the .global or .def 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.

Example

This example uses four files:

- `file1.lst` and `file3.lst` are equivalent. Both files define the symbol `Init` and make it available to other modules; both files use the external symbols `x`, `y`, and `z`. `file1.lst` uses the .global directive to identify these global symbols; `file3.lst` uses .ref and .def to identify the symbols.
- `file2.lst` and `file4.lst` are equivalent. Both files define the symbols `x`, `y`, and `z`, and make them available to other modules; both files use the external symbol `Init`. `file2.lst` uses the .global directive to identify these global symbols; `file4.lst` uses .ref and .def to identify the symbols.

file1.lst:

```
0001                        * Global symbol defined in this file
0002                                .global   Init
0003                        * Global symbols defined in another file
0004                                .global   x, y, z
0005 00000000               Init:
0006 00000000     09E0              MOVI      x, A0
     00000010 00000000!
0007                        ;         .
0008                        ;         .
0009                        ;         .
0010                                .end
```

file2.lst:

```
0001                        * Global symbols defined in this file
0002                                .global   x, y, z
0003                        * Global symbol defined in another file
0004                                .global   Init
0005                0001    x:      .set      1
0006                0002    y:      .set      2
0007                0003    z:      .set      3
0008 00000000       0000            .word     Init
0009                        ;         .
0010                        ;         .
0011                        ;         .
0012                                .end
```

file3.lst:

```
0001                        * Global symbol defined in this file
0002                                .def      Init
0003                        * Global symbols defined in another file
0004                                .ref      x, y, z
0005 00000000               Init:
0006 00000000     09E0              MOVI      x, A0
     00000010 00000000!
0007                        ;         .
0008                        ;         .
0009                        ;         .
0010                                .end
```

file4.lst:

```
0001                        * Global symbols defined in this file
0002                                .def      x, y, z
0003                        * Global symbol defined in another file
0004                                .ref      Init
0005                0001    x:      .set      1
0006                0002    y:      .set      2
0007                0003    z:      .set      3
0008 00000000       0000            .word     Init
0009                        ;         .
0010                        ;         .
0011                        ;         .
0012                                .end
```

Syntax

.if *well-defined 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 the 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 0 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    .set    1
0002                    0002  sym2    .set    2
0003                    0003  sym3    .set    3
0004                    0004  sym4    .set    4
0005
0006                          If_4:   .if     sym4 = sym2 * sym2
0007 00000000             04          .byte   sym4              ; Equal values
0008                                  .else
0009                                  .byte   sym2 * sym2       ; Unequal values
0010                                  .endif
0011
0012                          If_5:   .if       sym1 <= 10
0013 00000008             01          .byte   sym1              ; Less than/equal
0014                                  .endif
0015
0016                          If_6:   .if     sym3 * sym2 != sym4 + sym2
0017                                  .byte   sym4 + sym2       ; Unequal values
0018                                  .else
0019 00000010             06          .byte   sym3 * sym2       ; Equal values
0020                                  .endif
```

Syntax

.length *page length*

.width *page width*

Description

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

- Default length: 60 lines
- Minimum length: 1 line
- Maximum length: 32,767 lines

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

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

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

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

Example

This example shows source statements that change the page length and width.

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

             .length    65
             .width     85


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

             .length    55
             .width     100
```

Syntax **.nolist**

.list

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

The assembler does not print the .list or .nolist directives, or the directives that appear after a .nolist directive; however, it continues to increment the line counter. You can nest the .list/.nolist directives; each .nolist needs a matching .list to restore the listing. At the beginning of an assembly, the assembler acts as if it has assembled a .list directive.

> **Note:**
>
> If you don't request a listing file when you invoke the assembler, the assembler ignores the .list directive. To obtain a listing file, invoke the assembler with the -l (lowercase "L") option.

Example This example uses the .copy directive to insert source statements from another file. The first time this directive is encountered, the assembler lists the copied source lines in the listing file. The second time this directive is encountered, the assembler does not list the copied source lines because a .nolist directive was assembled. Note that the .nolist, the second .copy, and .list directives do not appear in the listing file; note also that the line counter is incremented even when source statements are not listed.

Source File:

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

Listing File:

```
 0001                                .copy    "copy2.asm"
A0001                          * In copy2.asm (copy file)
A0002 00000000 AABBCCDD              .long    0AABBCCDDh
 0002                          * Back in original file
 0007 00000040       44              .string  "Done"
      00000048       6F
      00000050       6E
      00000058       65
```

Syntax

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

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

Description

The .int and the .long directives are equivalent. They place one or more values into consecutive 32-bit fields in the current section. The operand field contains one or more *values* separated by commas. A *value* can be either:

- An expression which the assembler evaluates and treats as a 32-bit signed number.
- A character string enclosed in double quotes. Each character represents a separate value.

You can use up to 100 *values*, but the total line length cannot exceed 200 characters.

Example 1

This example uses the .long directive to initialize several 32-bit fields. The symbol `Dat1` points to the first reserved word.

```
0001 00000000 FFFFABCD  Dat1: .long  0FFFFABCDh, 'A' + 100h
     00000020 00000141
0002 00000040 00000000'        .long  Dat1, "long"
     00000060 0000006C
     00000080 0000006F
     000000A0 0000006E
     000000C0 00000067
```

Example 2

This example uses the .int directive to initialize several 32-bit fields. The symbol `X1` points to the first reserved word. Notice the difference between the character constant `'Inst'` and the character string `"Inst"`.

```
0001 00000000           X1:    .bss   page, 128, 16
0002 00000200 00000C80         .int   3200,page,1+'AB',X1
     00000220 00000000+
     00000240 00004242
     00000260 00000000+
0003 00000280 74736E49         .int   'Inst', "Inst"
     000002A0 00000049
     000002C0 0000006E
     000002E0 00000073
     00000300 00000074
```

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

(The quote marks surrounding the filename are optional.)

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

- Macro library members must be **source** files (not object files).
- The filename of a macro library member must be the same as the macro name and its extension must be .asm.

The *filename* must follow operating system conventions; it may be enclosed in double quotes. You can specify a full pathname (for example, `.mlib e:\gsp\macs.lib`). If you do not specify a full pathname, the assembler searches for the file in:

1) The directory that contains the current source file.
2) Any directories named with the -i assembler option.
3) Any directories specified by the environment variable A_DIR.

For more information about the -i option and the environment variable, see Section 4.3, Specifying Alternate Directories for Assembler Input, on page 4-4.

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 library members into the opcode table as library entries; this redefines any existing opcodes or macros that have the same name. If one of these macros is called, the assembler extracts the entry from the library and loads it into the macro table. The assembler expands the library entry in the same manner as other macros, but it does not place the source code into the listing. Only macros that are actually called from the library are extracted, and they are only extracted once.

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

inc_a0_8.asm

```
* Macro for incrementing
* A0 by 8
inc_a0_8 $MACRO
         ADDI    8, A0
         $ENDM
```

dec_a0_8.asm

```
* Macro for decrementing
* A0 by 8
dec_a0_8 $MACRO
         SUBI    8, A0
         $ENDM
```

Use the archiver to create a macro library:

```
gspar -a mac inc_a0_8.asm dec_a0_8.asm
```

Now you can use the .mlib directive to reference the macro library and define the `inc_a0_8.asm` and `dec_a0_8.asm` macros:

```
0002 00000000                        .mlib     "mac.lib"
0003 00000000     09C0               MOVI      9, A0
     00000010     0009
0004 00000020                        inc_A0_8
0001 00000020     0B00                 ADDI     8, A0
     00000030     0008
0005 00000040                        dec_A0_8
0001 00000040     0BE0                 SUBI     8, A0
     00000050     FFF7
```

Syntax **.mlist**

.mnolist

Description Two directives 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, all macro expansions are listed. The line counter restarts counting at 1 during a macro expansion; it resumes counting from its previous value when the macro expansion is complete. Unlisted macro expansion lines do not affect the line counter. Note that you can nest the .mlist and .mnolist directives.

Example This example defines a macro named `clr_a` that clears several registers in general-purpose register file A. The first time the macro is called, the macro expansion is listed (by default). The second time the macro is called, the macro expansion is not listed because a .mnolist directive was assembled. The third time the macro is called, the macro expansion is again listed because a .mlist directive was assembled.

```
0001                            clr_a       $MACRO
0002                                        CLR       A0
0003                                        CLR       A1
0004                                        CLR       A2
0005                                        CLR       A3
0006                                        $ENDM
0007
0008 00000000                               clr_a
0001 00000000      5600                     CLR       A0
0002 00000010      5621                     CLR       A1
0003 00000020      5642                     CLR       A2
0004 00000030      5663                     CLR       A3
0009 000000F0                               .mnolist
0010 000000F0                               clr_a
0011 000001E0                               .mlist
0012 000001E0                               clr_a
0001 000001E0      5600                     CLR       A0
0002 000001F0      5621                     CLR       A1
0003 00000200      5642                     CLR       A2
0004 00000210      5663                     CLR       A3
```

Syntax **.option** *option list*

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

B Limit the listing of .byte directives to one line.
D Limit the listing of .word directives to one line.
F Reset the B, D, L, M, and T options.
L Limit the listing of .long directives to one line.
M Turn of macro expansions in the listing.
T Limit the listing of .string directives to one line.
X Produce a symbol cross-reference listing.

Example This example limits the listings of the .byte, .word, .long, .int, and .string directives to one line each.

```
0001                                    ********************************************
0002                                    *  Limit the listing of .byte, .word, .int,  *
0003                                    *  .long, and .string directives to 1 line   *
0004                                    *  each                                      *
0005                                    ********************************************
0006 00000000                                   .option   B, D, L, T
0007 00000000          BD               byte:   .byte     -'C', 0B0h, 5
0008 00000020  AABBCCDD                 long:   .long     0AABBCCDDh, 536 + 'A'
0009 00000060       15AA                word:   .word     5546, 78h
0010 00000080  00000555                 int:    .int      010101010101b, 356q, 85
0011 000000E0          49               string: .string   "I/O Registers"
0012
0013                                    ********************************************
0014                                    *         Reset the listing options        *
0015                                    ********************************************
0016 00000148                                   .option   F
0017 00000148          BD                       .byte     -'C', 0B0h, 5
     00000150          B0
     00000158          05
0018 00000160  AABBCCDD                         .long     0AABBCCDDh, 536 + 'A'
     00000180  00000259
0019 000001A0       15AA                        .word     5546, 78h
     000001B0       0078
0020 000001C0  00000555                         .int      010101010101b, 356q, 85
     000001E0  000000EE
     00000200  00000055
0021 00000220          49                       .string   "I/O Registers"
     00000228          2F
     00000230          4F
     00000238          20
     00000240          52
     00000248          65
     00000250          67
     00000258          69
     00000260          73
     00000268          74
     00000270          65
     00000278          72
     00000280          73
```

Syntax **.page**

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

Example This example causes the assembler to begin new pages in the source listing.

Source file:

```
            .title    "*****  .page directive example  *****"
;             .
;             .
;             .
            .page
;             .
;             .
;             .
            .page
;             .
;             .
;             .
```

Listing file:

```
GSP COFF Assembler,  Version 1.30, 87.250              Tue Sep  1 08:51:08 1987
 (c) Copyright 1985, 1987, Texas Instruments Inc.

*****  .page directive example  *****                                  PAGE    1

 0002                       ;          .
 0003                       ;          .
 0004                       ;          .
GSP COFF Assembler,  Version 1.30, 87.250              Tue Sep  1 08:51:08 1987
 (c) Copyright 1985, 1987, Texas Instruments Inc.

*****  .page directive example  *****                                  PAGE    2

 0006                       ;          .
 0007                       ;          .
 0008                       ;          .
GSP COFF Assembler,  Version 1.30, 87.250              Tue Sep  1 08:51:08 1987
 (c) Copyright 1985, 1987, Texas Instruments Inc.

*****  .page directive example  *****                                  PAGE    3

 0010                       ;          .
 0011                       ;          .
 0012                       ;          .

 No Errors,  No Warnings
```

Syntax **.sect** *"section name"*

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

The *section name* identifies a section that the assembler assembles code into. The *name* is significant to 8 characters and must be enclosed in double quotes.

Section 3, Introduction to Common Object File Format, provides additional information about named sections.

Example This example defines two named sections, `init_grph` and `s_stk_1`, and assembles code into them.

```
0001
0002                4000  stk1_size:  .set    1024 * 16
0003                                  .global stk1_size
0004                0011  SPTCH       .set    B1
0005                0130  CONVSP      .set    0C00000130h
0006
0007                      ************************************************
0008                      * Begin assembling into a named section init_grph.  *
0009                      ************************************************
0010 00000000                         sect    "init_grph"
0011 00000000       0550              SETF    16, 0, 0
0012 00000010       1911              MOVK    8, SPTCH
0013
0014                      ************************************************
0015                      * Stop assembling into init_grph and begin assem-  *
0016                      * bling into the named section s_stk_1. This sec-  *
0017                      * tion reserves stk1_size bits; stk1_strt points   *
0018                      * to the beginning of the space, and stk1_end      *
0019                      * points to the end of the space.                  *
0020                      ************************************************
0021 00000000            stk1_strt:  .sect   "s_stk_1"
0022 00004000            stk1_end:   .bes    stk1_size
0023
0024                      ************************************************
0025                      * Stop assembling into s_stk_1 and begin assem-    *
0026                      * bling into .data. Equate STK_P1 with register    *
0027                      * A14; STK_P1 points to the beginning of the sec-  *
0028                      * tion s_stk_1.                                    *
0029                      ************************************************
0030 00000000                         .data
0031                0000E  STK_P1:     .set    A14
0032                                  .global STK_P1
0033 00000000       09EE              MOVI    stk1_strt, STK_P1
     00000010 00000000+
0034
0035                      ************************************************
0036                      * Stop assembling into .data and begin assembling  *
0037                      * into init_grph.                                  *
0038                      ************************************************
0039 00000020                         .sect   "init_grph"
0040 00000020       6A30              LMO     SPTCH, B0
0041 00000030       0590              MOVE    B0, @CONVSP, 0
     00000040 00000130
```

Syntax *symbol* **.set** *value*

Description The .set 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 assign meaningful names to 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 have been previously defined in the current 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 assigned to the symbol appears in the object field of the listing. This value is not part of the actual object code and is not written to the output file.

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

```
0001                          ****************************************
0002                          * Assign values to symbol names (define *
0003                          * assembly-time constants).            *
0004                          ****************************************
0005              0000  BLACK    .set     0
0006          11111111  BLUE     .set     011111111h
0007          22222222  RED      .set     022222222h
0008              0008  pix_sz   .set     8
0009
0010                          ****************************************
0011                          * Equate registers B8 and B9 with their *
0012                          * descriptive names and put values into *
0013                          * them.                                 *
0014                          ****************************************
0015              0018  COLOR0   .set     B8
0016              0019  COLOR1   .set     B9
0017 00000000     09F8           MOVI     BLUE, COLOR0
     00000010 11111111
0018 00000030     09F9           MOVI     RED, COLOR1
     00000040 22222222
0019
0020                          ****************************************
0021                          * Equate I/O register names with their  *
0022                          * memory addresses. Load the PSIZE reg- *
0023                          * ister with a value.                   *
0024                          ****************************************
0025          C0000150  PSIZE    .set     0C0000150h
0026          C0000160  PMASK    .set     0C0000160h
0027 00000060     1903           MOVK     pix_sz, A3
0028 00000070     0583           MOVE     A3, @PSIZE
     00000080 C0000150
0029
0030                          ****************************************
0031                          * Set symbol rel_sym to a relocatable   *
0032                          * expression and use it as a relocatable*
0033                          * operand.                              *
0034                          ****************************************
0035 000000A0     000A  label    .word    10
0036              00A1' rel_sym  .set     label + 1
0037 000000B0     0B23           ADDI     rel_sym, A3
     000000C0 000000A1'
```

Syntax **.space** *size in bits*

.bes *size in bits*

Description The .space and .bes directives reserve *size* number of bits in the current section and fill them with 0s. The section program counter is realigned to point to the bit following the reserved space.

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

> **Note:**
>
> The assembler performs no alignment before or after executing a .space directive. You can use the .even directive to align the SPC.

Example This example reserves memory with the .space and .bes directives.

```
0001                        ************************************************
0002                        **          Begin assembling into .text.      **
0003                        ************************************************
0004 00000000                       .text
0005 00000000     000A              .word  10
0006
0007                        ************************************************
0008                        ** Use the .space directive to reserve 12 bits in  **
0009                        ** the .text section. These bits are in the same   **
0010                        ** word whose first 4 bits were initialized by the **
0011                        ** .field directive. The label L1 points to bit    **
0012                        ** address 14h; this is the first bit reserved.    **
0013                        ************************************************
0014 00000010     0008              .field   8,4
0015 00000014               L1:     .space   12
0016 00000020     0014              .word    L1
0017
0018                        ************************************************
0019                        **          Begin assembling into .data.      **
0020                        ************************************************
0021 00000000                       .data
0022 00000000       44              .string  "Data"
     00000008       61
     00000010       74
     00000018       61
0023
0024                        ************************************************
0025                        ** Use the .space and .bes directives to reserve   **
0026                        ** 22 bits in the .data section. The first 8 bits  **
0027                        ** reserved by .space are in the same word whose   **
0028                        ** first 8 bits were initialized by a preceding    **
0029                        ** .byte directive. The label Res_1 points to bit  **
0030                        ** address 28h; this is the first bit reserved by  **
0031                        ** .space. The label Res_2 points to bit address   **
0032                        ** 4A; this is the last bit reserved by .bes.      **
0033                        ************************************************
0034 00000020       24              .byte    36
0035 00000028               Res_1:  .space   011h
0036 0000004A               Res_2:  .bes     011h
0037 00000050     0028              .word    Res_1
0038 00000060     004A              .word    Res_2
```

Syntax **.string** *"string$_1$" [, ...," string$_n$"]*

Description The .string directive places 8-bit characters from one or more character strings into consecutive bytes in the current section. Each *string* is either:

- An expression which the assembler evaluates and treats as a 16-bit signed number.
- A character string enclosed in double quotes. Each character in a string represents a separate byte.

The first character is aligned on an 8-bit boundary; the string is not padded.

Example This example initializes bytes with 8-bit ASCII characters.

```
0001 00000000                  .data
0002 00000000                  .space   1024
0003 00000400        65        .string  "end of data"
     00000408        6E
     00000410        64
     00000418        20
     00000420        6F
     00000428        66
     00000430        20
     00000438        64
     00000440        61
     00000448        74
     00000450        61
0004 00000000                  .sect    "strings"
0005 00000000        41  strs: .string  "Austin","San Antonio","Houston"
     00000008        75
     00000010        73
     00000018        74
     00000020        69
     00000028        6E
     00000030        53
     00000038        61
     00000040        6E
     00000048        20
     00000050        41
     00000058        6E
     00000060        74
     00000068        6F
     00000070        6E
     00000078        69
     00000080        6F
     00000088        48
     00000090        6F
     00000098        75
     000000A0        73
     000000A8        74
     000000B0        6F
     000000B8        6E
```

Syntax **.text**

Description The .text directive tells the assembler to begin assembling into the .text section, which contains executable code. The SPC is set to 0 if no code has been assembled into .text; if .text already contains code, the SPC is restored to its previous value in the section.

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

For more information about COFF sections, see Section 3.

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

```
0001                    0000  PSIZE     .set   0C0000150
0002                    0000  PMASK     .set   0C0000160
0003                          ************************************
0004                          **   Begin assembling into .text   **
0005                          ************************************
0006 00000000                           .text
0007 00000000           0360            DINT
0008 00000010           0550            SETF   16, 0, 0
0009 00000020           0744            SETF   4, 0, 1
0010
0011                          ************************************
0012                          **   Begin assembling into .data   **
0013                          ************************************
0014 00000000                           .data
0015 00000000           0000  plmask    .word  0000
0016 00000010           0004  pixsz     .word  4
0017
0018                          ************************************
0019                          **   Resume assembling into .text  **
0020                          ************************************
0021 00000030                           .text
0022 00000030           05C0            MOVE   @plmask, @PMASK
     00000040 00000000"
     00000060 00000000
0023 00000080           05C0            MOVE   @pixsz, @PSIZE
     00000090 00000010"
     000000B0 00000000
```

Syntax **.title** *"string"*

.title *'string'*

(Enclose the string with single or double quotes.)

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

The *string* is a quote-enclosed title of up to 65 characters. If you supply more than 65 characters, the assembler truncates the string and issues a warning. The assembler prints the title on the page that follows the directive, and on subsequent pages until another .title directive is processed.

Example This example prints the title`**** Floating-Point Routines ****` on the first listing page and `**** Pixel Block Transfers ****` on the second page.

Source file:

```
        .title   "**** Floating-Point Routines ****"
;          .
;          .
;          .
        .page
        .title   "**** Pixel Block Transfers ****"
;          .
;          .
;          .
```

Listing file:

```
GSP COFF Assembler,  Version 1.30, 87.250              Wed Sep  9 16:50:53 1987
 (c) Copyright 1985, 1987, Texas Instruments Inc.

**** Floating-Point Routines ****                                    PAGE    1

 0002                    ;          .
 0003                    ;          .
 0004                    ;          .

GSP COFF Assembler,  Version 1.30, 87.250              Wed Sep  9 16:50:53 1987
 (c) Copyright 1985, 1987, Texas Instruments Inc.

**** Pixel Block Transfers ****                                      PAGE    2

 0006                    ;          .
 0007                    ;          .
 0008                    ;          .
```

Syntax *symbol* **.usect** *"section name", size in bits [, word alignment flag]*

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

- The *symbol* points to the first location reserved by this invocation of the .usect directive. The *symbol* corresponds to the name of the variable that you're reserving space for.
- The *section name* must be enclosed in double quotes; only the first 8 characters are significant. This parameter names the uninitialized section.
- The *size* is an expression that defines the number of bits that are reserved in section *name*.
- The *word alignment flag* is an optional parameter. If you specify a nonzero value for the *alignment flag*, the assembler aligns the reserved space on a 16-bit (word) boundary within .bss. If you specify a value of zero for the *alignment flag*, the assembler does not align the reserved space; this is also the default if *alignment flag* is **not** specified.

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

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

For more information about COFF sections, see Section 3.

Example This example uses the .usect directive to define two uninitialized, named sections, `pixarray` and `uninit_V`. The symbol `ar_1` points to the first bit reserved in the `pixarray` section. The symbol `ar_2` points to the first bit in a block of 400 bits reserved in `pixarray`. The symbol `VEC` points to the first bit reserved in the `uninit_V` section.

Figure 5-11 shows how this example reserves space in the uninitialized sections.

```
0001                        ****************************************
0002                        *      Begin assembling into .text     *
0003                        ****************************************
0004 00000000                       .text
0005 00000000     5600              CLR     A0
0006
0007                        ****************************************
0008                        * Reserve 400 bits in the named section *
0009                        * pixarray. Space is reserved for a var-*
0010                        * iable named ar_1; ar_1 is an array of *
0011                        * 100 4-bit pixels. The space is aligned*
0012                        * on a word boundary.                   *
0013                        ****************************************
0014 00000000           ar_1    .usect  "pixarray", 100 * 4, 1
0015
0016 00000010     0B00              ADDI    36, A0        ; Still in .text
     00000020     0024
0017
0018                        ****************************************
0019                        * Reserve 400 additional bits in section*
0020                        * pixarray. This time, the space is re- *
0021                        * served for a variable named ar_2.     *
0022                        ****************************************
0023 00000190           ar_2    .usect  "pixarray", 100 * 4 , 1
0024
0025 00000030     0B00              ADDI    48, A0        ; Still in .text
     00000040     0030
0026
0027                        ****************************************
0028                        * Reserve 32 bits in the named section  *
0029                        * uninit_V for the variable VEC.        *
0030                        ****************************************
0031 00000000           VEC     .usect  "uninit_V", 32, 1
0032
0033 00000050     4C02              MOVE    A0, A2        ; Still in .text
0034
0035                        ****************************************
0036                        *   Declare .usect symbols as external  *
0037                        ****************************************
0038                              .global ar_1, ar_2, VEC
```

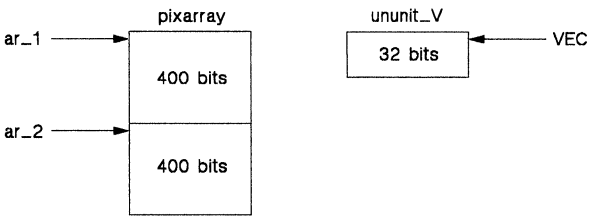


Figure 5-11. An Example of the .usect Directive

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

Description The .word directive places one or more values into consecutive 16-bit words in the current section. The assembler evaluates each expression and places the value in a word as a 16-bit number. You can use as many values as fit on a single line.

> **Note:**
>
> If the *value* is a relocatable expression, the assembler truncates it to a 16-bit expression without warning.

Example This example initializes five words. The symbol WORDX points to the first word.

```
0001 00000000     0C80  WORDX:   .word    3200,1+'AB',-'AF',0F4A0h,'A'
     00000010     4242
     00000020     B9BF
     00000030     F4A0
     00000040     0041
```

Section 6

Instruction Set

The TMS34010 supports a base set of general-purpose instructions as well as special-purpose instructions that are particularly suited for graphics applications.

This section does not cover topics such as opcodes or instruction timing; the *TMS34010 User's Guide* discusses the instruction set in detail. The *TMS34010 User's Guide* also contains an alphabetical presentation which is similar to the directives reference that begins on page 5-14.

This section provides a general summary of the TMS34010 instruction set. Section 6.1 provides an overview of the TMS34010 operand formats and describes symbols that are used throughout this section. Section 6.2 lists the syntax, operation, and description of each instruction, Section 6.3 through Section 6.9 describe the functional categories of the instruction set.

Section **Page**

6.1 Overview of Operand Formats

The TMS34010 instruction set supports eight categories of operand formats. Most instructions have register-direct operands or a combination of register-direct and immediate operands. The MOVE, MOVB, and graphics instructions, however, use more complex combinations of operands.

This section describes the symbols that are used in the instruction syntaxes to show operands formats.

- **Immediate Values and Constants**

  An instruction syntax may use one of these symbols to indicate an immediate *source* operand:

  IW is a 16-bit (short) signed immediate value.
  IL is a 32-bit (long) signed immediate value.
  K is a 5-bit constant.

  Instructions that have immediate source operands have register-direct destination operands. Many instructions that have an immediate value can use either a short or a long value. Here's an example of an instruction that uses an immediate operand:

```
ADDI  36h, A0
```

  This example adds an immediate value, 36h, to the contents of register A0.

- **Absolute Addresses**

  An instruction syntax may use one of these symbols to indicate an absolute operand:

  @SAddress is a **source** address that contains the source data.
  @DAddress is a **destination** address.

  Note that the @ character is entered as part of the operand (this distinguishes it from an immediate operand). Here are two examples of instructions that use absolute operands:

```
MOVB  @loop, A4
MOVE  B2, @100h+2h
```

  The first instruction moves a byte from the address specified by the symbol `loop` into register A4. The second instruction moves the contents of register B2 into address 102h.

- **Register-Direct Operands**

  An instruction syntax may use one of these symbols to indicate a register-direct operand:

  Rs is a **source** register that contains the source data.
  Rd is a **destination** register that will contain the result.

When both operands of an instruction are register-direct operands, the registers *must be in the same file.* (The MOVE Rs, Rd instruction is an exception to this rule.) Here's an instruction that uses two register-direct operands:

```
ADD    A0, A4
```

This instruction adds the contents of register A0 to the contents of register A4 and leaves the result in A4.

- **Register-Indirect Operands**

An instruction syntax may use one of these symbols to indicate a register-indirect operand:

Rs* is a register that contains the address of the **source data.
Rd* is a register that contains the **destination address.

Note that the * character is entered as part of the operand (this distinguishes it from a register-direct operand). Here's an instruction that uses two register-indirect operands:

```
MOVE   *A0, *A4
```

Register A0 contains an address and register A4 contains an address. This instruction moves the data from the address specified by A0 into the address specified by A4.

- **Register-Indirect with Offset**

An instruction syntax may use one of these symbols to indicate a register-indirect operand that uses a signed offset:

Rs(offset)* is a **source address formed by adding an offset to the contents of the source register.
Rd(offset)* is a **destination address formed by adding an offset to the contents of the destination register.

The offset is only used to form an address – the contents of the register are not affected.

Note that the * character is entered as part of the operand. Here's an example that uses a register-indirect source operand with a displacement:

```
MOVE   A4, *A3(16)
```

This instruction moves the data in register A4 to a destination address. The destination address is formed by adding the offset, 16, to the contents of register A3.

- **Register-Indirect with Postincrement**

An instruction syntax may use one of these symbols to indicate a register-indirect operand that is postincremented:

Rs+* is a register that contains the address of the **source data.
Rd+* is a register that contains the **destination address.

After the operation is performed, the contents of the specified source or destination register are incremented by the field size used for the operation.

Note that the * and + characters are entered as part of the operand. Here's an example that uses an indirect destination operand with postincrement:

```
MOVE   A4, *A5+, 1
```

This instruction moves the contents of register A4 to the address specified by the contents of A5. The third operand, 1, indicates that FS1 and FE1 should be used for this move. After the data is moved, the contents of FS1 are added to the contents of A5.

- **Register-Indirect with Predecrement**

An instruction syntax may use one of these symbols to indicate a register-indirect operand that is predecremented.

Before the operation is performed, the contents of the specified source or destination register are decremented by the field size used for the operation.

-Rs* the decremented register contents are the address of the **source data.
-Rd* the decremented register contents specify the **destination address.

Note that the * and - characters are entered as part of the operand. Here's an example that uses an indirect source operand with predecrement:

```
MOVE   *-A4, A5, 0
```

The third operand of this instruction, 0, indicates that FS0 and FE0 should be used for this move. Before the move is performed, the contents of FS0 are subtracted from the contents of register A4 to produce the source address. The contents of this address are moved into register A5.

- **Register-Indirect in XY Mode**

An instruction syntax may use one of these symbols to indicate that the a register operands contains an XY address.

Rs*.XY** is a register that contains the XY address of the **source** data.
Rd*.XY** is a register that contains the XY **destination** address.

Note that the * and **.XY** characters are entered as part of the operand. Here's an example that uses an indirect-XY destination operand:

```
PIXT   A0, *A6.XY
```

This instruction moves the contents of register A0 into the XY address specified by the contents of register A6.

6.2 Summary Table

Syntax	Description
ABS *Rd*	**Store Absolute Value** Operation: |Rd| → Rd Store the absolute value of the specified register back into the register.
ADD *Rs, Rd*	**Add Registers** Operation: Rs + Rd → Rd Add the contents of the source register to the contents of the destination register.
ADDC *Rs, Rd*	**Add Registers with Carry** Operation: Rs + Rd + C → Rd Add the contents of the source register and the carry bit to the contents of the destination register.
ADDI *IW, Rd, [**W**]* **ADDI** *IL, Rd, [**L**]*	**Add Immediate - *Short or Long*** Operation: immediate value + Rd → Rd Add an immediate value to the contents of the destination register. In the short form, the operand is a 16-bit sign-extended value. In the long form, the operand is a 32-bit signed value. You can force the assembler to use the short (16-bit) form of this instruction by using the **W** operand. You can force the assembler to use the long (32-bit) form of this instruction by using the **L** operand.
ADDK *K, Rd*	**Add Constant (5 Bits)** Operation: K + Rd → Rd Add a 5-bit constant to the contents of the destination register. The constant K is treated as an unsigned number in the range 1–32; K=32 is converted to 0 in the opcode. The assembler issues an error if you try to add 0 to a register.
ADDXY *Rs, Rd*	**Add Registers in XY Mode** Operation: Rs**X** + Rd**X** → Rd**X** Rs**Y** + Rd**Y** → Rd**Y** The registers are signed XY registers. Add the X half of the source register to the X half of the destination register, and add the Y half of the source register to the Y half of the destination register. All the values are signed. Any carry out from the X portion does not propagate into the Y portion.
AND *Rs, Rd*	**AND Registers** Operation: Rs AND Rd → Rd Bitwise-AND the contents of the source register with the contents of the destination register.

Key:

Rs - Source register
RsX, RdX - X half (16 LSBs) of Rs or Rd
SAddress - 32-bit source address
IW - 16-bit (short) immediate value
Address - 32-bit address (label)
K - 5-bit constant
PC' - Next instruction
Rd - Destination register
RsY, RdY - Y half (16 MSBs) of Rs or Rd
DAddress - 32-bit destination address
IL - 32-bit (long) immediate value
F - Field select; defaults to 0
F=0 selects FS0 and FE0
F=1 selects FS1 and FE1

Syntax	Description
ANDI *IL, Rd*	**AND Immediate (32 Bits)** Operation: IL AND Rd → Rd Bitwise-AND a 32-bit immediate value with the contents of the destination register.
ANDN *Rs, Rd*	**AND Register with Complement** Operation: (NOT Rs) AND Rd → Rd Bitwise-AND the 1s complement of the source register contents with the contents of the destination register.
ANDNI *IL, Rd*	**AND Not Immediate (32 Bits)** Operation: (NOT IL) AND Rd → Rd Bitwise-AND the 1s complement of a 32-bit immediate value with the contents of the destination register.
BTST *K, Rd*	**Test Register Bit - *Constant*** Operation: set status on value of bit K in Rd Test bit K in the destination register, and set the Z bit. If K=0, set Z to 1; if K=1, set Z to 0.
BTST *Rs, Rd*	**Test Register Bit - *Register*** Operation: Rs specifies a bit number; set status on value of this bit in Rd Rs specifies the bit number to test. Test this bit in the destination register, and set the Z bit. If the bit is 0, set Z to 1; if the bit is 1, set Z to 0.
CALL *Rs*	**Call Subroutine - *Indirect*** Operation: PC' → TOS Rs → PC SP -32 → SP Rs specifies the address of a subroutine. The assembler pushes the address of the next instruction (PC') onto the stack, then jumps to address of the subroutine. Note that when Rs=SP, Rs is decremented *after* it is written to the PC (the PC contains the original value of Rs).
CALLA *Address*	**Call Subroutine - *Absolute*** Operation: PC' → TOS *Address* → PC The assembler pushes the address of the next instruction (PC') onto the stack, then jumps to the specified 32-bit address.
CALLR *Address*	**Call Subroutine - *Relative*** Operation: PC' → TOS PC' + (displacement × 16) → PC The assembler pushes the address of the next instruction (PC') onto the stack, then jumps to the address specified by PC' + displacement. The address must be within the current section.
CLR *Rd*	**Clear Register** Operation: Rd XOR Rd → Rd Set the contents of the register to 0.
CLRC	**Clear Carry** Operation: 0 → C Set the contents of the carry bit to 0.

Syntax	Description
CMP *Rs, Rd*	**Compare Registers** Operation: set status bits on the result of Rd - Rs Set the status bits as if the contents of the source register were subtracted from the contents of the destination register. *This is a nondestructive compare* – the register contents do not change.
CMPI *IW, Rd, [**W**]* **CMPI** *IL, Rd, [**L**]*	**Compare Immediate – *Short or Long*** Operation: set status bits on the result of Rd - immediate value Set the status bits as if an immediate value were subtracted from the contents of the destination register. *This is a nondestructive compare* – the register contents do not change. CMPI is used with conditional jumps. In the short form, the operand is a 16-bit sign-extended value. In the long form, the operand is a 32-bit signed value. You can force the assembler to use the short (16-bit) form of this instruction by using the **W** operand. You can force the assembler to use the long (32-bit) form of this instruction by using the **L** operand.
CMPXY *Rs, Rd*	**Compare X and Y Halves of Registers** Operation: set status bits on the results of: Rd**X** - Rs**X** Rd**Y** - Rs**Y** The registers are signed XY registers. Set the status bits as if Rs**X** were subtracted from Rd**X** and Rs**Y** were subtracted from Rd**Y**. No overflow detection is provided. *This is a nondestructive compare* – the register contents do not change.
CPW *Rs, Rd*	**Compare Point to Window** Operation: point code → Rd The source register contains an XY address. The assembler compares this address to the window limits defined by the WSTART and WEND registers. Bits 5–8 of the destination register are set to a 4-bit code that identifies the location of the specified address with respect to the window. The following diagram shows the codes in relation to the window; the upper left corner is at address [0,0]. 0101 | 0100 | 0110 0001 | 0000 (window) | 0010 1001 | 1000 | 1010
CVXYL *Rs, Rd*	**Convert XY Address to Linear Address** Operation: XY address in Rs → linear address in Rd Convert the XY address in the source register to a linear address that is placed in the destination register.

Key:

Rs – Source register
RsX, RdX – X half (16 LSBs) of Rs or Rd
SAddress – 32-bit source address
IW – 16-bit (short) immediate value
Address – 32-bit address (label)
K – 5-bit constant
PC' – Next instruction
Rd – Destination register
RsY, RdY – Y half (16 MSBs) of Rs or Rd
DAddress – 32-bit destination address
IL – 32-bit (long) immediate value
F – Field select; defaults to 0
F=0 selects FS0 and FE0
F=1 selects FS1 and FE1

Syntax	Description
DEC *Rd*	**Decrement Register** Operation: Rd - 1 → Rd Subtract 1 from the contents of the destination register.
DINT	**Disable Interrupts** Operation: 0 → IE Disable interrupts by setting the global interrupt enable bit (IE) to 0.
DIVS *Rs, Rd*	**Divide Registers - *Signed*** Operation: Rd even: Rd:Rd+1 / Rs → Rd Remainder → Rd+1 Rd odd: Rd / Rs → Rd Perform a signed divide. There are two cases: **Rd even:** The contents of Rd and the contents of the next consecutive register (Rd+1) form a 64-bit operand. Rd specifies the 32 MSBs and Rd+1 specifies the 32 LSBs. The 64-bit operand is divided by the contents of the source register. The result is stored in Rd, and the remainder is stored in Rd+1. The remainder and the result always have the same sign. **Rd odd:** Divide the contents of the destination register by contents of the source register. The destination register contains the result; the remainder is not stored.
DIVU *Rs, Rd*	**Divide Registers - *Unsigned*** Operation: Rd even: Rd:Rd+1 / Rs → Rd Remainder → Rd+1 Rd odd: Rd / Rs → Rd Perform an unsigned divide. There are two cases: **Rd even:** The contents of Rd and the contents of the next consecutive register (Rd+1) form a 64-bit operand. Rd specifies the 32 MSBs and Rd+1 specifies the 32 LSBs. The 64-bit operand is divided by the contents of the source register. The result is stored in Rd, and the remainder is stored in Rd+1. The remainder and the result always have the same sign. **Rd odd:** Divide the contents of the destination register by contents of the source register. The destination register contains the result; the remainder is not stored.
DRAV *Rs, Rd*	**Draw and Advance** Operation: COLOR1 pixel value → *Rd Rs**X** + Rd**X** → Rd**X** Rs**Y** + Rd**Y** → Rd**Y** Write the pixel value in the COLOR1 register to the XY address contained in Rd, then increment the X half of Rd by RsX and the Y half of Rd by RsY.

Syntax	Description
DSJ *Rd, Address* **DSJS** *Rd, Address*	**Decrement Register and Skip Jump – *Short or Long*** Operation: Rd - 1 → Rd If Rd ≠ 0, then (displacement × 16) + PC' → PC If Rd = 0, then go to next instruction Decrement the contents of the destination register by 1. If the result is **nonzero**, then jump to the address specified by (displacement × 16) + PC'. If the result is **0**, then skip the jump and continue execution at PC'. This instruction has a short form and a long form. The assembler automatically chooses the short form if the displacement is 5 bits or less, which provides a jump range of ±32 words (excluding 0). The assembler automatically chooses the long form if the displacement is greater than 5 bits, which provides a jump range of -32,768 to +32,767.
DSJEQ *Rd, Address*	**Conditionally Decrement Register and Skip Jump** Operation: If Z = 1: Rd - 1 → Rd If Rd ≠ 0, then (displacement × 16) + PC' → PC If Rd = 0, then go to next instruction If Z = 0 go to next instruction If **Z=1**, subtract 1 from the contents of the destination register. If the result is *nonzero*, then jump to the address specified by (displacement × 16) + PC'. If the result is *0*, then skip the jump and continue execution at PC'. If **Z=0**, skip the jump and continue execution at PC'.
DSJNE *Rd, Address*	**Conditionally Decrement Register and Skip Jump** Operation: If Z = 0: Rd - 1 → Rd If Rd ≠ 0, then (displacement × 16) + PC' → PC If Rd = 0, then go to next instruction If Z = 1 go to next instruction If **Z=0**, subtract 1 from the contents of the destination register. If the result is *nonzero*, then jump to the address specified by (displacement × 16) + PC'. If the result is *0*, then skip the jump and continue execution at PC'. If **Z=1**, skip the jump and continue execution at PC'.
EINT	**Enable Interrupts** Operation: 1 → IE Set the global interrupt enable bit (IE) to 1. Individual interrupts can now be enabled by setting the appropriate bits in the INTENB register.

Key:

Rs – Source register
RsX, RdX – X half (16 LSBs) of Rs or Rd
SAddress – 32-bit source address
IW – 16-bit (short) immediate value
Address – 32-bit address (label)
K – 5-bit constant
PC' – Next instruction
Rd – Destination register
RsY, RdY – Y half (16 MSBs) of Rs or Rd
DAddress – 32-bit destination address
IL – 32-bit (long) immediate value
F – Field select; defaults to 0
F=0 selects FS0 and FE0
F=1 selects FS1 and FE1

Syntax	Description
EMU	**Initiate Emulation** Operation: ST → Rd conditionally enter emulator mode Pulse the $\overline{\text{EMUA}}$ pin and sample the RUN/$\overline{\text{EMU}}$ pin. If RUN/$\overline{\text{EMU}}$ is in the RUN state, the EMU instruction acts as a NOP. If the pin is in the EMU state, the processor enters emulation mode.
EXGF *Rd [, F]*	**Exchange Field Size** Operation: Rd → FS0, FE0 **or** Rd → FS1, FE1 FS0, FE0 → Rd **or** FS1, FE1 → Rd Exchange the 6 LSBs of the destination register with the selected 6 bits of field information. *F* is an optional parameter that can be 0 or 1. If you specify 0, then FS0 and FE0 are selected; if you specify 1, then FS1 and FE1 are selected. FS0 and FE0 are the default. Bit 5 of the 5-bit value in Rd is exchanged with the field extension value. The 26 MSBs of Rd are cleared.
EXGPC *Rd*	**Exchange Program Counter** Operation: Rd → PC PC' → Rd Exchange the next program counter value with the contents of the destination register. After EXGPC completes executing, Rd contains the address of the instruction that immediately follows the EXGPC.
FILL L	**Fill Array with Processed Pixels – *Linear*** Operation: COLOR1 pixel values → pixel array (with processing) Combine an array of source pixels (specified by COLOR1) with a destination pixel array, based on the selected graphics operations. The address of the destination array is a *linear* address.
FILL XY	**Fill Array with Processed Pixels – *XY*** Operation: COLOR1 pixel values → pixel array (with processing) Combine an array of source pixels (specified by COLOR1) with a destination pixel array, based on the selected graphics operations. The address of the destination array is an *XY* address.
GETPC *Rd*	**Get Program Counter into Register** Operation: PC' → Rd Increment the PC to point to the next instruction, and copy that address into the destination register.
GETST *Rd*	**Get Status Register into Register** Operation: ST → Rd Copy the contents of the status register into the destination register.
INC *Rd*	**Increment Register** Operation: PC + 1 → Rd Add 1 to the contents of the destination register.
JA*cc Address*	**Jump Absolute Conditional** Operation: If *cc* = true, then Address → PC If *cc* = false, then go to next instruction If the condition is **true**, jump to the address. If the condition is **false**, continue execution at PC'. *cc* is one of the condition codes listed in Table 6-11 on page 6-33.

Syntax	Description
JR*cc Address*	**Jump Relative Conditional** Operation: If *cc* = true, then displacement + PC' → PC If *cc* = false, then go to next instruction If the condition is **true**, jump to the address specified by displacement + PC'. If the condition is **false**, continue execution at PC'. *cc* is one of the condition codes listed in Table 6-11 on page 6-33. This instruction has a short form and a long form. The assembler automatically chooses the short form if the displacement is 8 bits or less; this provides a jump range of ±127 words excluding 0). The assembler automatically chooses the long form if the displacement is greater than 8 bits; this provides a jump range of ±32K words excluding 0).
JUMP *Rs*	**Jump Indirect** Operation: Rs → PC Jump to the address specified by the source register.
LINE *[0 , 1]*	**Line Draw with XY Addressing** Operation: **LINE 0** While COUNT > 0 Draw the next pixel If $d \geq 0$ $d = d + 2b - 2a$ POINTER = POINTER + INC1 Else $d = d + 2b$ POINTER = POINTER + INC2 **LINE 1** While COUNT > 0 Draw the next pixel If $d > 0$ $d = d + 2b - 2a$ POINTER = POINTER + INC1 Else $d = d + 2b$ POINTER = POINTER + INC2 Perform the inner loop of Bresenham's line-drawing algorithm.
LMO *Rs, Rd*	**Leftmost One** Operation: 31 - bit number of leftmost 1 in Rs → Rd Locate the leftmost 1 in the source register. Load the 1s complement of the *bit number* of the leftmost one into the 5 LSBs of the destination register. The 27 MSBs of Rd are zeroed.

Key:

Rs – Source register
RsX, RdX – X half (16 LSBs) of Rs or Rd
SAddress – 32-bit source address
IW – 16-bit (short) immediate value
Address – 32-bit address (label)
K – 5-bit constant
PC' – Next instruction
Rd – Destination register
RsY, RdY – Y half (16 MSBs) of Rs or Rd
DAddress – 32-bit destination address
IL – 32-bit (long) immediate value
F – Field select; defaults to 0
F=0 selects FS0 and FE0
F=1 selects FS1 and FE1

Syntax	Description
MMFM *Rs [, reg. list]*	**Move Multiple Registers from Memory** Operation: If Register *n* is in the register list, *Rs+ → R*n* (repeat for *n* = 0 to 15) Load the contents of a specified list of either A or B registers (not both) from a block of memory. Rs points to the first location in the memory block; Rs and the registers in the list must be in the same file. If you don't specify a register list, **all** the registers that are in the same file as Rs are moved.
MMTM *Rd [, reg. list]*	**Move Multiple Registers to Memory** Operation: If Register *n* is in the register list, R*n* → -*Rd (repeat for *n* = 0 to 15) Store the contents of a specified list of either A or B registers (not both) into a block of memory. Rd points to the first location in the memory block; Rd and the registers in the list must be in the same file. If you don't specify a register list, **all** the registers that are in the same file as Rd are moved.
MODS *Rs, Rd*	**Modulus - *Signed*** Operation: Rd mod Rs → Rd Divide the contents of the destination register by the contents of the source register, and load the remainder into the destination register. The contents of the registers are treated as signed numbers.
MODU *Rs, Rd*	**Modulus - *Unsigned*** Operation: Rd mod Rs → Rd Divide the contents of the destination register by the contents of the source register, and load the remainder into the destination register. The contents of the registers are treated as signed numbers.
MOVB *Rs, *Rd*	**Move Byte - *Register to Indirect*** Operation: Rs → *Rd Move the least significant byte of the source register to a memory address. The destination address is specified by the contents of the destination register.
MOVB *Rs, *Rd(offset)*	**Move Byte - *Register to Indirect with Offset*** Operation: Rs → *(Rd + offset) Move the least significant byte of the source register to memory. The memory address is formed by adding the offset to the contents of the destination register.
MOVB *Rs, @DAddress*	**Move Byte - *Register to Absolute*** Operation: Rs → @DAddress Move the least significant byte of the source register to the specified memory address.
MOVB **Rs, Rd*	**Move Byte - *Indirect to Register*** Operation: *Rs → Rd Move a byte from a memory location into the destination register; the byte is right-justified and sign-extended within Rd. The source address is specified by the contents of the source register.

Syntax	Description
MOVB **Rs, *Rd*	**Move Byte – *Indirect to Indirect*** Operation: *Rs → *Rd Move a byte from a memory location into another memory location. The source address is specified by the contents of the source register, and the destination address is specified by the contents of the destination register.
MOVB **Rs(offset), Rd*	**Move Byte – *Indirect with Offset to Register*** Operation: *(Rs + offset) → Rd Move a byte from a memory location into the destination register; the byte is right-justified and sign-extended within Rd. The source address is formed by adding the offset to the contents of the source register.
MOVB **Rs(offset), *Rd(offset)*	**Move Byte – *Indirect with Offset to Indirect with Offset*** Operation: *(Rs + offset) → *(Rd + offset) Move a byte from a memory location into another memory location. The source address is formed by adding an offset to the contents of the source register, and the destination address is formed by adding an offset to the contents of the destination register.
MOVB *@SAddress, Rd*	**Move Byte – *Absolute to Register*** Operation: @SAddress → Rd Move a byte from a memory location to the destination register.
MOVB *@SAddress, @DAddress*	**Move Byte – *Absolute to Absolute*** Operation: @SAddress → @DAddress Move a byte from a memory location into another memory location.
MOVE *Rs, Rd*	**Move – *Register to Register*** Operation: Rs → Rd Move the contents of the source register into the destination register. *It is not necessary for the registers to be in the same file.*
MOVE *Rs, *Rd [, F]*	**Move Field – *Register to Indirect*** Operation: field in Rs → field in *Rd Move a field from the source register into a memory location specified by the contents of the destination register.
MOVE *Rs, *Rd+ [, F]*	**Move Field – *Register to Indirect (Postincrement)*** Operation: field in Rs → field in *Rd Rd + field size → Rd Move a field from the source register into a memory location specified by the contents of the destination register. After the move, increment the contents of Rd by the field size.

Key:

Rs – Source register
RsX, RdX – X half (16 LSBs) of Rs or Rd
SAddress – 32-bit source address
IW – 16-bit (short) immediate value
Address – 32-bit address (label)
K – 5-bit constant
PC' – Next instruction
Rd – Destination register
RsY, RdY – Y half (16 MSBs) of Rs or Rd
DAddress – 32-bit destination address
IL – 32-bit (long) immediate value
F – Field select; defaults to 0
F=0 selects FS0 and FE0
F=1 selects FS1 and FE1

Syntax	Description
MOVE *Rs, -*Rd [, F]*	**Move Field – *Register to Indirect (Predecrement)*** Operation: Rd - field size → Rd field in Rs → field in *Rd Decrement the contents of Rd by the field size. Move a field from the source register into a memory location specified by the contents of the destination register.
MOVE *Rs, *Rd(offset) [, F]*	**Move Field – *Register to Indirect with Offset*** Operation: field in Rs → field in *(Rd + offset) Move a field from the source register into a memory location. The destination address is formed by adding an offset to the contents of the destination register.
MOVE *Rs, @DAddress [, F]*	**Move Field – *Register to Absolute*** Operation: field in Rs → field in memory Move a field from the source register into the specified destination address.
MOVE **Rs, Rd [, F]*	**Move Field – *Indirect to Register*** Operation: field in *Rs → field in Rd Move a field from contents of a memory address into the destination register. The source address is specified by the contents of the source register.
MOVE **Rs, *Rd [, F]*	**Move Field – *Indirect to Indirect*** Operation: field in *Rs → field in *Rd Move a field from the contents of a memory address into another memory address. The source address is specified by the contents of the source register, and the destination address is specified by the contents of the destination register.
MOVE **Rs+, Rd [, F]*	**Move Field – *Indirect (Postincrement) to Register*** Operation: field in *Rs → field in Rd Rs + field size → Rs Move a field from the contents of a memory address into the destination register. The source address is specified by the contents of the source register. After the move, increment the contents of the source register by the field size.
MOVE **Rs+, *Rd+ [, F]*	**Move Field – *Indirect (Postincrement) to Indirect (Postincrement)*** Operation: field in *Rs → field in *Rd Rs + field size → Rs Rd + field size → Rd Move a field from the contents of a memory address into another memory address. The source address is specified by the contents of the source register, and the destination address is specified by the contents of the destination register. After the move, increment the contents of both the source register and the destination register by the field size.
MOVE *-*Rs, Rd [, F]*	**Move Field – *Indirect (Predecrement) to Register*** Operation: Rs - field size → Rs field in *Rs → field in Rd Decrement the contents of the source register by the field size. Move a field from the contents of a memory address into the destination register. The source address is specified by the contents of the source register.

Syntax	Description
MOVE -**Rs, -**Rd [, F]*	**Move Field – *Indirect (Predecrement) to Indirect (Predecrement)*** Operation: Rs - field size → Rs Rd - field size → Rd field in *Rs → field in *Rd Decrement the contents of both the source register and the destination register by the field size. Move a field from the contents of a memory address into another memory address. The source address is specified by the contents of the source register, and the destination address is specified by the contents of the destination register.
MOVE **Rs*(*offset*), *Rd [, F]*	**Move Field – *Indirect with Offset to Register*** Operation: field in *(Rs + offset) → field in Rd Move a field from the contents of a memory address into the destination register. The source address is formed by adding an offset to the contents of the source register.
MOVE **Rs*(*offset*), **Rd+ [, F]*	**Move Field – *Indirect with Offset to Indirect (Postincrement)*** Operation: field in *(Rs + offset) → field in *Rd Move a field from the contents of a memory address into the destination register. The source address is formed by adding an offset to the contents of the source register. After the move, increment the contents of the destination register by the field size.
MOVE **Rs*(*offset*), **Rd*(*offset*) [, F]*	**Move Field – *Indirect with Offset to Indirect with Offset*** Operation: field in *(Rs + offset) → field in *(Rd + offset) Move a field from the contents of a memory address into another memory address. The source address is formed by adding an offset to the contents of the source register, and the destination address is formed by adding an offset to the contents of the destination register.
MOVE *@SAddress, Rd [, F]*	**Move Field – *Absolute to Register*** Operation: field in source address → field in Rd Move a field from the contents of the specified memory address into the destination register.
MOVE *@SAddress, **Rd+ [, F]*	**Move Field – *Absolute to Indirect(Postincrement)*** Operation: field in source address → field in *Rd Rd + field size → Rd Move a field from the contents of the specified source address into the memory address specified by the contents of the destination register. After the move, increment the contents of the destination register by the field size.

Key:

Rs – Source register
RsX, RdX – X half (16 LSBs) of Rs or Rd
SAddress – 32-bit source address
IW – 16-bit (short) immediate value
Address – 32-bit address (label)
K – 5-bit constant
PC' – Next instruction
Rd – Destination register
RsY, RdY – Y half (16 MSBs) of Rs or Rd
DAddress – 32-bit destination address
IL – 32-bit (long) immediate value
F – Field select; defaults to 0
F=0 selects FS0 and FE0
F=1 selects FS1 and FE1

Syntax	Description
MOVE *@SAddress, @DAddress [, F]*	**Move Field – *Absolute to Absolute*** Operation: field in source address → field in destination address Move a field from the contents of the specified source address into the specified destination address.
MOVI *IW, Rd, [**W**]* **MOVI** *IL, Rd, [**L**]*	**Move Immediate – *Short or Long*** Operation: immediate operand → Rd Move an immediate value into the destination register. In the short form, the operand is a 16-bit sign-extended value. In the long form, the operand is a 32-bit signed value. You can force the assembler to use the short (16-bit) form of this instruction by using the **W** operand. You can force the assembler to use the long (32-bit) form of this instruction by using the **L** operand.
MOVK *K, Rd*	**Move Constant – *5 Bits*** Operation: K → Rd Move a 5-bit constant into the destination register. Note that this instruction does not affect the status register.
MOVX *Rs, Rd*	**Move X Half of Register** Operation: Rs**X** → Rd**X** Move the X half (16 LSBs) of the source register into the X half of the destination register. The Y portions are not affected.
MOVY *Rs, Rd*	**Move Y Half of Register** Operation: Rs**Y** → Rd**Y** Move the Y half (16 LSBs) of the source register into the Y half of the destination register. The X portions are not affected.
MPYS *Rs, Rd*	**Multiply Registers – *Signed*** Operation: Rd even: Rs × Rd → Rd:Rd+1 Rd odd: Rs × Rd → Rd Perform a signed multiply of a field in the source register by the 32-bit contents of the destination register. This produces a 32-bit to a 64-bit result, depending on the register and the field definition.
MPYU *Rs, Rd*	**Multiply Registers – *Unsigned*** Operation: Rd even: Rs × Rd → Rd:Rd+1 Rd odd: Rs × Rd → Rd Perform an unsigned multiply of a field in the source register by the 32-bit contents of the destination register. This produces a 32-bit to a 64-bit result, depending on the register and the field definition.
NEG *Rd*	**Negate Registers** Operation: -Rd → Rd Store the 2s complement of the contents of the destination register back into the destination register.
NEGB *Rd*	**Negate Registers with Borrow** Operation: -Rd - C → Rd Take the 2s complement of the contents of the destination register; if the carry bit is set, decrement the result by 1.

Syntax	Description
NOP	**No Operation** Operation: no operation Increment the program counter to point to the next instruction.
NOT *Rd*	**Complement Register** Operation: NOT Rd → Rd Store the 1s complement of the contents of the destination register back into the destination register.
OR *Rs, Rd*	**OR Registers** Operation: Rs OR Rd → Rd Bitwise-OR the contents of the source register with the contents of the destination register.
ORI *IL, Rd*	**OR Immediate** Operation: IL OR Rd → Rd Bitwise-OR a 32-bit immediate value with the contents of the destination register.
PIXBLT B, L	**Pixel Block Transfer – *Binary to Linear*** Operation: binary source pixel array → destination pixel array (with processing) Expand, transfer, and process a source pixel array with a destination pixel array according to the selected graphics operations. The starting addresses for both arrays are linear addresses. The source array contains 1-bit pixels.
PIXBLT B, XY	**Pixel Block Transfer – *Binary to XY*** Operation: binary source pixel array → destination pixel array (with processing) Expand, transfer, and process a source pixel array with a destination pixel array according to the selected graphics operations. The starting address of the source array is a linear address; the starting address of the destination array is an XY address. The source array contains 1-bit pixels.
PIXBLT L, L	**Pixel Block Transfer – *Linear to Linear*** Operation: source pixel array → destination pixel array (with processing) Transfer and process a source pixel array with a destination pixel array according to the selected graphics operations. The starting addresses of both arrays are linear addresses.
PIXBLT L, XY	**Pixel Block Transfer – *Linear to XY*** Operation: source pixel array → destination pixel array (with processing) Transfer and process a source pixel array with a destination pixel array according to the selected graphics operations. The starting address of the source array is a linear address; the starting address of the destination array is an XY address.

Key:

Rs – Source register
RsX, RdX – X half (16 LSBs) of Rs or Rd
SAddress – 32-bit source address
IW – 16-bit (short) immediate value
Address – 32-bit address (label)
K – 5-bit constant
PC' – Next instruction
Rd – Destination register
RsY, RdY – Y half (16 MSBs) of Rs or Rd
DAddress – 32-bit destination address
IL – 32-bit (long) immediate value
F – Field select; defaults to 0
F=0 selects FS0 and FE0
F=1 selects FS1 and FE1

Syntax	Description
PIXBLT XY, L	**Pixel Block Transfer – *XY to Linear*** Operation: source pixel array → destination pixel array (with processing) Transfer and process a source pixel array with a destination pixel array according to the selected graphics operations. The starting address of the source array is an XY address; the starting address of the destination array is a linear address.
PIXBLT XY, XY	**Pixel Block Transfer – *XY to XY*** Operation: source pixel array → destination pixel array (with processing) Transfer and process a source pixel array with a destination pixel array according to the selected graphics operations. The starting addresses of the both arrays are XY addresses.
PIXT *Rs, *Rd*	**Pixel Transfer – *Register to Indirect*** Operation: pixel in Rs → pixel in *Rd Transfer a pixel from the source register to the linear address specified by the contents of the destination register.
PIXT *Rs, *Rd*.XY	**Pixel Transfer – *Register to Indirect XY*** Operation: pixel in Rs → pixel in *Rd.XY Transfer a pixel from the source register to the XY address specified by the contents of the destination register.
PIXT **Rs, Rd*	**Pixel Transfer – *Indirect to Register*** Operation: pixel in *Rs → pixel in Rd Transfer a pixel from a linear address to the destination register. The source address is specified by the contents of the source register.
PIXT **Rs, *Rd*	**Pixel Transfer – *Indirect to Indirect*** Operation: pixel in *Rs → pixel in *Rd Transfer a pixel from a linear address to another linear address. The source address is specified by the contents of the source register, and the destination address is specified by the contents of the source destination register.
PIXT **Rs*.XY, *Rd*	**Pixel Transfer – *Indirect XY to Register*** Operation: pixel in *Rs.XY → pixel in Rd Transfer a pixel from an XY address to the destination register. The source address is specified by the contents of the source register.
PIXT **Rs*.XY, **Rd*.XY	**Pixel Transfer – *Indirect XY to Indirect XY*** Operation: pixel in *Rs.XY → pixel in *Rd.XY Transfer a pixel from an XY address to another XY address. The source address is specified by the contents of the source register, and the destination address is specified by the contents of the source destination register.
POPST	**Pop Status Register from Stack** Operation: *SP+ → ST Move the two words on the top of the stack into the status register. After the move, increment the SP by 32.

Syntax	Description
PUSHST	**Push Status Register on Stack** Operation: ST → -*SP Decrement the SP by 32, then move the contents of the status register onto the top of the stack.
PUTST *Rs*	**Copy Register into Status** Operation: Rs → ST Copy the contents of the source register into the status register.
RETI	**Return from Interrupt** Operation: *SP+ → ST *SP+ → PC Return from an interrupt routine. Pop the status register and the program counter from the stack and continue executing from the address in the PC.
RETS *[N]*	**Return from Subroutine** Operation: *SP → PC (N defaults to 0) SP + 32 + 16N → SP Return from a subroutine. Move the program counter from the stack and increment SP by 32 + 16N; if N is not specified, increment the SP by 32.
REV *Rd*	**Store Revision Number** Operation: revision number → Rd Load the revision number of the TMS34010 into the destination register.
RL *K, Rd*	**Rotate Left - *Constant*** Operation: Rd rotated left by K → Rd Left-rotate the contents of the destination register by the value of K (K specifies a value between 0–31).
RL *Rs, Rd*	**Rotate Left - *Register*** Operation: Rd rotated left by Rs → Rd Left-rotate the contents of the destination register by the value in the 5 LSBs of the source register. (The 5 LSBs of the source register specify a value between 0–31; the 27 MSBs are ignored.)
SETC	**Set Carry** Operation: 1 → C Set the carry bit to 1.
SETF *FS, FE [, F]*	**Set Field Parameters** Operation: (FS, FE) → ST Load the values specified for the field size and field extension bits into the status register. The remainder of the status register is not affected.
SEXT *Rd [, F]*	**Sign Extend to Long** Operation: field in Rd → sign-extended field in Rd Sign extend a field in the destination register by copying the MSB of the field into the nonfield bits.

Key:

Rs - Source register
RsX, RdX - X half (16 LSBs) of Rs or Rd
SAddress - 32-bit source address
IW - 16-bit (short) immediate value
Address - 32-bit address (label)
K - 5-bit constant
PC' - Next instruction
Rd - Destination register
RsY, RdY - Y half (16 MSBs) of Rs or Rd
DAddress - 32-bit destination address
IL - 32-bit (long) immediate value
F - Field select; defaults to 0
F=0 selects FS0 and FE0
F=1 selects FS1 and FE1

Syntax	Description
SLA *K, Rd*	**Shift Left Arithmetic - *Constant*** Operation: Left-shift Rd by K → Rd Left-shift the contents of the destination register by the value of K. (K specifies a value between 0–31.) 0s are left-shifted into the LSBs and the last bit shifted out is shifted into the carry bit. If either the sign bit (N) or any bits shifted out of Rd differ from the original sign bit, the overflow (V) bit is set.
SLA *Rs, Rd*	**Shift Left Arithmetic - *Register*** Operation: left-shift Rd by Rs → Rd Left-shift the contents of the destination register by the value in the 5 LSBs of the source register. (The 5 LSBs of the source register specify a value between 0–31; the 27 MSBs are ignored.) 0s are left-shifted into the LSBs and the last bit shifted out is shifted into the carry bit. If either the sign bit (N) or any bits shifted out of Rd differ from the original sign bit, the overflow (V) bit is set.
SLL *K, Rd*	**Shift Left Logical - *Constant*** Operation: left-shift Rd by K → Rd Left-shift the contents of the destination register by the value of K. (K specifies a value between 0–31.) 0s are left-shifted into the LSBs and the last bit shifted out is shifted into the carry bit. Note that this instruction does not affect the overflow bit.
SLL *Rs, Rd*	**Shift Left Logical - *Register*** Operation: left-shift Rd by Rs → Rd Left-shift the contents of the destination register by the value in the 5 LSBs of the source register. (The 5 LSBs of the source register specify a value between 0–31; the 27 MSBs are ignored.) 0s are left-shifted into the LSBs and the last bit shifted out is shifted into the carry bit. Note that this instruction does not affect the overflow bit.
SRA *K, Rd*	**Shift Right Arithmetic - *Constant*** Operation: right-shift Rd by K → Rd Right-shift the contents of the destination register by the value of K. (K specifies a value between 0–31.) The original value of the sign bit (MSB) is shifted into the MSBs of Rd, the last bit shifted out is shifted into the carry bit.
SRA *Rs, Rd*	**Shift Right Arithmetic - *Register*** Operation: right-shift Rd by Rs → Rd Right-shift the contents of the destination register by the 2s complement value of the 5 LSBs in the source register. (The 5 LSBs of the source register specify a value between 0–31; the 27 MSBs are ignored.) The original value of the sign bit (MSB) is shifted into the MSBs of Rd, the last bit shifted out is shifted into the carry bit.
SRL *K, Rd*	**Shift Right Logical - *Constant*** Operation: right-shift Rd by K → Rd Right-shift the contents of the destination register by the value of K. (K specifies a value between 0–31.) 0s are shifted into the MSBs of Rd, and the last bit shifted out is shifted into the carry bit.
SRL *Rs, Rd*	**Shift Right Logical - *Register*** Operation: right-shift Rd by Rs → Rd Right-shift the contents of the destination register by the 2s complement value of the 5 LSBs in the source register. (The 5 LSBs of the source register specify a value between 0–31; the 27 MSBs are ignored.) 0s are shifted into the MSBs of Rd, and the last bit shifted out is shifted into the carry bit.

Syntax	Description
SUB *Rs, Rd*	**Subtract Registers** Operation: Rd - Rs → Rd Subtract the contents of the source register from the contents of the destination register.
SUBB *Rs, Rd*	**Subtract Registers with Borrow** Operation: Rd - Rs - C → Rd Subtract the contents of the source register and the carry bit from the contents of the destination register.
SUBI *IW, Rd, [W]* **SUBI** *IL, Rd, [L]*	**Subtract Immediate – *Short or Long*** Operation: Rd - immediate value → Rd Subtract an immediate value from the contents of the destination register. In the short form, the operand is a 16-bit sign-extended value. In the long form, the operand is a 32-bit signed value. You can force the assembler to use the short (16-bit) form of this instruction by using the **W** operand. You can force the assembler to use the long (32-bit) form of this instruction by using the **L** operand.
SUBK *K, Rd*	**Subtract Constant** Operation: Rd - K → Rd Subtract a 5-bit constant from the contents of the destination register. *K* is an unsigned number in the range 1–32; *K*=0 in the opcode corresponds to the value 32.
SUBXY *Rs, Rd*	**Subtract Registers in XY Mode** Operation: Rd**X** - Rs**X** → Rd**X** Rd**Y** - Rs**Y** → Rd**Y** Subtract the X half of Rs from the X half of Rd and subtract the Y half of Rs from the Y half of Rd. The destination register contains the result.
TRAP *N*	**Software Interrupt** Operation: PC → -*SP ST → -*SP trap vector *N* → PC Perform a software interrupt. *N* is a trap vector in the range 0–31. The return address (the address of the next instruction) and the status register are pushed onto the stack, interrupts are disabled, ST is set to 00000010h, and the trap vector is loaded into the PC.
XOR *Rs, Rd*	**Exclusive-OR Registers** Operation: Rs XOR Rd → Rd Bitwise-exclusive-OR the contents of the source register with the contents of the destination register.
XORI *IL, Rd*	**Exclusive-OR Immediate Value** Operation: IL XOR Rd → Rd Bitwise-exclusive-OR a 32-bit immediate value with the contents of Rd.
ZEXT *Rd [, F]*	**Zero-Extend to Long** Operation: field in Rd → zero-extended field in Rd Zero-extend a field in the destination register by setting all the non-field bits to 0.

Key:

Rs – Source register
RsX, RdX – X half (16 LSBs) of Rs or Rd
SAddress – 32-bit source address
IW – 16-bit (short) immediate value
Address – 32-bit address (label)
K – 5-bit constant
PC' – Next instruction

Rd – Destination register
RsY, RdY – Y half (16 MSBs) of Rs or Rd
DAddress – 32-bit destination address
IL – 32-bit (long) immediate value
F – Field select; defaults to 0
F=0 selects FS0 and FE0
F=1 selects FS1 and FE1

6.3 Arithmetic, Logical, and Compare Instructions

The TMS34010 supports a full range of arithmetic, logical, and compare instructions. Most of these instructions use register-direct operands; some use a combination of immediate and register-direct operands. Some instructions have several versions; each uses a different operand format. For example, the *ADD* instruction has several versions:

- The **ADD** instruction uses register-direct operands for both the source and destination operands.
- The **ADDI** instruction uses an immediate source with a destination register.
- The **ADDK** instruction uses a 5-bit constant as the source operand with a destination register.
- The **ADDXY** instruction is similar to the ADD instruction – both operands are register-direct operands – however, the registers contain XY values.

Some instructions that have immediate values as source operands (such as the ADDI instruction) have two forms: a *short form* and a *long form*. In the short form, the source operand is a *16-bit* immediate value and the instruction occupies *two words*. In the long form, the source operand is a *32-bit* immediate value and the instruction occupies *three words*. Each form of the instruction has an optional third operand: **W** for short and **L** for long. If you don't use the **W** or **L** operand, the assembler chooses the short or the long form, depending on the size of the source operand. Using **W** or **L** *forces* the assembler to use the short or long form, respectively. If you use **W** and the source value is greater than 16 bits, the assembler discards all but the 16 LSBs and issues a warning message. If you use **L** and the source value is less than 32 bits, the assembler sign-extends the value to 32 bits.

Some instructions that use immediate operands have only one version. In this case, the operand is long (32-bits).

> **Note:**
>
> When an instruction's source and destination operands are both register-direct operands, the registers *must be in the same file.* (The MOVE Rs, Rd instruction is an exception to this rule.)

Table 6-1 summarizes the arithmetic, logical, and compare instructions.

Table 6-1. Summary of Arithmetic, Logical, and Compare Instructions

Instruction	Description	Instruction	Description
ABS *Rd*	Store absolute value	**LMO** *Rs, Rd*	Locate leftmost one
ADD *Rs, Rd*	Add registers	**MODS** *Rs, Rd*	Modulus (signed)
ADDC *Rs, Rd*	Add registers with carry	**MODU** *Rs, Rd*	Modulus (unsigned)
ADDI *IW, Rd* [,**W**] **ADDI** *IL, Rd* [,**L**]	Add immediate (short or long)	**MPYS** *Rs, Rd*	Multiply registers (signed)
ADDK *K, Rd*	Add constant	**MPYU** *Rs, Rd*	Multiply registers (unsigned)
ADDXY *Rs, Rd* ‡	Add registers in XY mode	**NEG** *Rd*	Negate register
AND *Rs, Rd*	AND registers	**NEGB** *Rd*	Negate register with borrow
ANDI *IL, Rd*	AND immediate (long only)	**NOT** *Rd*	Complement register
ANDN *Rs, Rd*	AND register with complement	**OR** *Rs, Rd*	OR registers
ANDNI *IL, Rd*	AND immediate with complement (long only)	**ORI** *IL, Rd*	OR immediate (long only)
BTST *Rs, Rd* † **BTST** *K, Rd*	Test register bit (register or constant)	**SETC**	Set carry
CLR *Rd*	Clear register	**SEXT** *Rd* [, F]	Sign extend to long
CLRC	Clear carry bit	**SUB** *Rs, Rd*	Subtract registers
CMP *Rs, Rd*	Compare registers	**SUBB** *Rs, Rd*	Subtract registers with borrow
CMPI *IW, Rd* [,**W**] **CMPI** *IL, Rd* [,**L**]	Compare immediate (short or long)	**SUBI** *IW, Rd* [,**W**] **SUBI** *IL, Rd* [,**L**]	Subtract immediate (short or long)
CMPXY *Rs, Rd* ‡	Compare registers in XY mode	**SUBK** *K, Rd*	Subtract constant
DEC *Rd*	Decrement register	**SUBXY** *Rs, Rd* ‡	Subtract registers in XY mode
DIVS *Rs, Rd*	Divide registers (signed)	**XOR** *Rs, Rd*	Exclusive-OR registers
DIVU *Rs, Rd*	Divide registers (unsigned)	**XORI** *IL, Rd*	Exclusive-OR immediate (long only)
INC *Rd*	Increment register	**ZEXT** *Rd* [, F]	Zero-extend to long

† This instruction has a register version and a constant version; both versions use the same mnemonic.
‡ Section 6.9 discusses XY instructions.

6.4 Move Instructions

The TMS34010 supports a variety of move instructions, allowing you to move immediate values into registers, move data between registers, and move data between registers and memory. Table 6-2 summarizes the move instructions.

Table 6-2. Summary of Move Instructions

Instruction	Description	Instruction	Description
MMFM *Rs [, reg. list]*	Move multiple registers from memory	**MOVI** *IW, Rd* [, **W**] † **MOVI** *IL, Rd* [, **L**]	Move immediate
MMTM *Rs [, reg. list]*	Move multiple registers to memory	**MOVK** *K, Rd*	Move constant
MOVB ‡	Move byte	**MOVX** *Rs, Rd* ¶	Move X half of register
MOVE §	Move field	**MOVY** *Rs, Rd* ¶	Move Y half of register

† Section 6.3 discusses immediate instructions that have both short (16-bit) and long (32-bit) versions.
‡ Nine versions (see Table 6-4)
§ Eighteen versions (see Table 6-3)
¶ Section 6.9 discusses XY instructions.

- The **MMTM** and **MMFM** instructions use register-direct operands. MMTM allows you to save several register values in memory; MMFM allows you to retrieve register values from memory. The *Rs* operand for these instructions is a pointer; it contains the address where the register values are stored or obtained. The *reg. list* operand is an optional list of registers. It specifies which registers are stored or retrieved, and also indicates the storing or retrieval order. Note that Rs and all the registers in the list *must be in the same register file*. If you do not specify a register list, the entire register file is stored in or retrieved from memory; the register file selected depends on which register file Rs is in.

- The **MOVI** and **MOVK** instructions move immediate values into registers. The MOVI instruction has two forms; see Section 6.3 (page 6-22) for a description of this type of immediate instruction.

- The **MOVX** and **MOVY** instructions move values into the 16 LSBs or 16 MSBs, respectively, of a register. See Section 6.9 (page 6-33) for a discussion of XY instructions.

- The **MOVE** instruction supports eighteen combinations of operand formats. There are four basic combinations:

  - Register to register,
  - Register to memory,
  - Memory to register, **and**
  - Memory to memory.

  The MOVE instruction moves a *field*. A field is a configurable data structure that is identified by its starting address and its length. Field lengths can range from 1 to 32 bits. A field's memory address points to the LSB of the field; the field occupies contiguous bits. A field in a register is right-justified within the register; the field's LSB coincides with the register's LSB.

Note that all forms of the MOVE instruction (except *MOVE Rs, Rd*) have an optional **F** parameter. **F** selects the field size and field extension for the MOVE. If F=0, FS0 and FE0 determine the field size and extension. If F=1, FS1 and FE1 determine the field size and extension. If you don't specify 0 or 1, 0 is used as the default.

Table 6-3 summarizes the valid combinations of operand formats for the MOVE instruction.

Table 6-3. Summary of Operand Formats for the MOVE Instruction

	Destination					
Source	*Rd*	**Rd*	**Rd+*	*-***Rd*	**Rd(offset)*	*@DAddress*
Rs	√	√	√	√	√	√
**Rs*	√	√				
**Rs+*	√		√			
*-***Rs*	√			√		
**Rs(offset)*	√		√		√	
@SAddress	√		√			√

- The **MOVB** instruction is a special form of the MOVE instruction; when you use MOVB, the field size is restricted to 8 bits. MOVB supports nine combinations of operand formats. There are three basic combinations:

  - Register to memory,
  - Memory to register, **and**
  - Memory to memory.

  (Note that the MOVB instruction does not move data between registers.)

  The MOVB instruction allows a byte to begin on any bit boundary in memory. The byte's memory address points to the LSB of the byte. When a byte is moved into a register, the byte's LSB coincides with the register's LSB; the byte is sign-extended into the 24 MSBs of the register.

  Table 6-4 summarizes the valid combinations of operand formats for the MOVB instruction.

Table 6-4. Summary of Operand Formats for the MOVB Instruction

	Destination			
Source	*Rd*	**Rd*	**Rd(offset)*	*@DAddress*
Rs		√	√	√
**Rs*	√	√		
**Rs(offset)*	√		√	
@SAddress	√			√

6.5 Graphics Instructions

The TMS34010 instruction set supports several fundamental graphics drawing operations. Table 6-5 summarizes the graphics instructions. (Note that the *TMS34010 User's Guide* contains detailed descriptions of graphics operations, pixel blocks, graphics registers, and other related topics.)

Table 6-5. Summary of Graphics Instructions

Instruction	Description	Instruction	Description
CPW *Rs, Rd*	Compare point to window	**FILL XY** †	Fill array with processed pixels (XY)
CVXYL *Rs, Rd* †	Convert XY address to linear address	**LINE** [**0, 1**]	Line draw with XY addressing
DRAV *Rs, Rd*	Draw and advance	**PIXBLT** ‡	Pixel block transfer
FILL L	Fill array with processed pixels (linear)	**PIXT** §	Pixel transfer

† Six versions (see Table 6-7)
‡ Six versions (see Table 6-6)
§ Section 6.9 discusses XY instructions.

- The **CPW** instruction compares a point to the window limits defined by the WSTART and WEND registers. The source operand *Rs* contains an XY address. After the compare operation is performed, bits 5–8 contain a code that indicate the point's location with respect to the window limits. The entry for the CPW instruction in Section 6.2 shows these point codes.

- The **CVXYL** instruction converts an XY address to a 32-bit linear address. The source register contains the XY address; the linear address is put in the destination register.

- The **DRAV** instruction draws the pixel value in the COLOR1 register to the XY address specified by the destination register. After the pixel is drawn, the Y half of Rs is added to the Y half of Rd, and the X half of Rs is added to the X half of Rd.

- The **LINE** instruction performs the inner loop of Bresenham's line-drawing algorithm to draw a diagonal, horizontal, or vertical line. The optional operand may be a 0 or a 1; this selects one of two algorithms. The default for this operand is 0.

- The **FILL** instruction fills a two-dimensional pixel array with the value in the COLOR1 register. Note that **L** and **XY** *are not operands*; they are part of the instruction mnemonic, identifying the form of the FILL instruction. FILL L specifies that the array has a linear starting address; FILL XY specifies that the array has an XY starting address.

- The **PIXT** instruction transfers a pixel from one location to another. PIXT can transfer a pixel:
  - From a register to memory,
  - From memory to a register, **or**
  - From memory to memory.

  Table 6-6 summarizes the valid combinations of operand formats for the PIXT instruction. Note that all addresses are linear unless the operand is suffixed with **.XY**.

Table 6-6. Summary of Operand Formats for the PIXT Instruction

	Destination Pixel		
Source Pixel	*Rd*	**Rd*	**Rd*.XY
Rs		√	√
**Rs*	√	√	
**Rs*.XY	√		√

- The **PIXBLT** instruction moves a two-dimensional block of pixels from one memory location to another. Note that **B**, **L**, and **XY** *are not operands*; instead, they identify the source or destination array starting addresses as binary, linear, or XY addresses. The source and destination addresses of the arrays are designated by the SADDR and DADDR registers, respectively.

  Table 6-7 summarizes the various combinations of pixel block transfers.

Table 6-7. Summary of Array Types for the PIXBLT Instruction

	Destination Array	
Source Array	**Linear**	**XY**
Binary	√	√
Linear	√	√
XY	√	√

The graphics instructions use the B-file registers and several I/O registers as *implied operands*. These registers must be loaded with appropriate values *before the instruction is executed*. The TMS34010 obtains information from these registers during instruction execution. Table 6-8 summarizes the implied operands that are used by the graphics instructions. The *TMS34010 User's Guide* contains a complete discussion of these registers and describes the types of information they should contain.

Note that registers B10–B13 are temporary registers for most instructions; for the LINE instruction, however, these registers have the following functions:

B10: COUNT register
B11: INC1 register
B12: INC2 register
B13: PATTRN register

Table 6-8. Immediate Operands Used by Graphics Instructions

	B File Registers															I/O Registers				
	SADDR	SPTCH	DADDR	DPTCH	OFFSET	WSTART	WEND	DYDX	COLOR0	COLOR1	B10	B11	B12	B13	B14	CONTROL	CONVSP	CONVDP	PSIZE	PMASK
CPW *Rs, Rd*						XY	XY													
CVXYL *Rs, Rd*				L	L													√	√	
DRAV *Rs, Rd*				L	L	XY	XY			P						(1)		√	√	√
FILL L			L	L				XY		P	√	√	√	√	√	(2)			√	√
FILL XY			XY†	L	L	XY	XY	XY†		P	√	√	√	√	√	(1)		√	√	√
LINE [**0, 1**]	I		XY		L	XY	XY	XY		P	I	XY	XY	pat	√	(1)		√	√	√
PIXBLT B, L	L	L	L	L				XY	P	P	√	√	√	√	√	(2)			√	√
PIXBLT B, XY	L	L	XY†	L	L	XY	XY	XY†	P	P	√	√	√	√	√	(1)	√	√	√	√
PIXBLT L, L	L†	L	L†	L				XY			√	√	√	√	√	(4)	(5)	(5)	√	√
PIXBLT L, XY	L	L	XY†	L	L	XY	XY	XY†			√	√	√	√	√	(3)	√	√	√	√
PIXBLT XY, L	XY	L	L	L	L			XY			√	√	√	√	√	(4)	√	√	√	√
PIXBLT XY, XY	XY	L	XY†	L	L	XY	XY	XY†			√	√	√	√	√	(3)	√	√	√	√
PIXT *Rs, *Rd*																(2)			√	√
PIXT *Rs, *Rd*.**XY**				L	L	XY	XY									(1)		√	√	√
PIXT **Rs, Rd*																			√	√
PIXT **Rs, *Rd*																(2)			√	√
PIXT **Rs*.**XY**, *Rd*				L	L												√		√	√
PIXT **Rs*.**XY** , **Rd*.**XY**		L		L	L	XY	XY									(1)	√	√	√	√

Key:
▒ Changed by instruction execution
√ Used; no particular format
XY Register is in XY format
L Register is in linear format
P Register is in pixel format
pat Register is in pattern format
† Changed as a result of common rectangle function with window hit operation (W=1)
(1) CONTROL bits used: PP, W, T
(2) CONTROL bits used: PP, T
(3) CONTROL bits used: PP, W, T, PBH, PBV
(4) CONTROL bits used: PP, T, PBH, PBV
(5) Used when PBV=1

6.6 Program Control and Context Switching Instructions

The TMS34010 supports the ability to:

- Call and return from subroutines,
- Enable or disable interrupts,
- Set software interrupts, **and**
- Set, save, or restore status information.

Most of these instructions use register-direct or absolute operands; however, several of them have no operands.

Table 6-9 summarizes these instructions.

Table 6-9. Summary of Program Control and Context Switching Instructions

Instruction	Description	Instruction	Description
CALL *Rs*	Call subroutine (indirect)	**POPST**	Pop status register from stack
CALLA *Address*	Call subroutine (absolute)	**PUSHST**	Push status register onto stack
CALLR *Address*	Call subroutine (relative)	**PUTST**	Copy a register's contents into ST
DINT	Disable interrupts	**RETI**	Return from interrupt (immediate)
EINT	Enable interrupts	**RETS** *[N]*	Return from subroutine
GETPC *Rd*	Get PC into register	**REV** *Rd*	Store revision number
GETST *Rd*	Get ST into register	**SETF** *FS, FE [, F]*	Set field size and extension
NOP	No operation	**TRAP** *N*	Software interrupt

6.7 Jump Instructions

The TMS34010 supports both conditional and unconditional jumps. The conditional jumps use absolute operands or a combination of register-direct and absolute operands.

Table 6-10. Summary of Jump Instructions

Instruction	Description	Instruction	Description
DSJ *Rd, Address*	Decrement register and skip jump	**JA***cc* *Address*	Jump absolute conditional
DSJEQ *Rd, Address*	Conditionally decrement register and skip jump	**JR***cc* *Address* †	Jump relative conditional
DSJNE *Rd, Address*	Conditionally decrement register and skip jump	**JUMP** *Rs*	Jump indirect
DSJS *Rd, Address*	Decrement register and skip jump (short)		

† This instruction has a short (±127 words) version and a long (±32K words) version; both versions use the same mnemonic.

- As Table 6-10 shows, there are four DSJ instructions:
  - **DSJ** and **DSJS** decrement the contents of a register and jump to the specified address if the new contents of Rd do not equal 0. If Rd is decremented to 0, then execution continues with the next instruction.

    DSJ provides a jump range of -32,768 to +32,767 words; DSJS provides a jump range of ±32 words (excluding 0).
  - The operation of **DSJEQ** and **DSJNE** depends on the value of the Z (zero) status bit.

    DSJEQ decrements the contents of Rd when **Z=1** and jumps to the specified address if the new contents of Rd do not equal 0. If Rd is decremented to 0, then execution continues with the next instruction. If **Z=0**, DSJEQ skips the jump and execution continues with the next instruction.

    DSJNE decrements the contents of Rd when **Z=0** and jumps to the specified address if the new contents of Rd do not equal 0. If Rd is decremented to 0, then execution continues with the next instruction. If **Z=0**, DSJNE skips the jump and execution continues with the next instruction.

  The address specified for the DSJ instructions is relative; the assembler uses this address automatically to calculate a displacement, and then it inserts the displacement into the instruction.

- The **JUMP** instruction is unconditional. The source register contains the address for the jump.

- The conditional jump instructions, **JA***cc* and **JR***cc*, use the condition codes listed Table 6-11.

  The JR*cc* instruction has a long and a short form. The short form supports a jump range of ±127 words (excluding 0). The long form supports a jump range of ±32K words (excluding 0).

The 32-bit address specified for the JA*cc* instruction is absolute; the assembler inserts this address into words 2 and 3 of the instruction. The address specified for the JR*cc* instructions is relative; the assembler uses this address automatically to calculate a displacement, and then it inserts the displacement into the instruction. The short form has an 8-bit displacement that is inserted into bits 0–7 of the opcode; the opcode is 1 word long. The long form has 16-bit displacement; the opcode is 2 words long, and the displacement occupies the entire 16 bits of the second word.

Table 6-11 lists the condition codes used with the JR*cc* and JA*cc* instructions. (To use the codes, replace the *cc* with the appropriate mnemonic code; for example, JRUC, JALS, JRYGT, etc.) Before using one of these jump instructions, use the CMP, CMPI, or CMPXY instruction; the compare instructions set the condition codes for the jump by subtracting a source value from a destination value. The first mnemonics code column in Table 6-11 lists the codes that can be used for a jump following a CMP or CMPI. The second mnemonics code column list codes that can be used for a jump following a CMPXY (codes that are preceded with an *X* can be used with the result of the X comparison and codes that are preceded with a *Y* can be used with the result of the Y comparison).

Table 6-11. Condition Codes for JR*cc* and JA*cc* Instructions

	Mnemonic Code		Result of Compare	Status Bits	Code
Unconditional Compares	UC	–	Unconditional	don't care	0000
Unsigned Compares	LO (C)	–	Dst lower than Src	C	0001
	LS	YLE	Dst lower or same as Src	$\underline{C} + \underline{Z}$	0010
	HI	YGT	Dst higher than Src	$\overline{C} \cdot \overline{Z}$	0011
	HS (NC)	–	Dst higher or same as Src	$\overline{C}$	1001
	EQ (Z)	–	Dst = Src	$\underline{Z}$	1010
	NE (NZ)	–	Dst ≠ Src	$\overline{Z}$	1011
Signed Compares	LT	XLE	Dst < Src	$(N \cdot \overline{V}) + (\overline{N} \cdot V)$	0100
	LE	–	Dst ≤ Src	$(N \cdot \overline{V} + (\overline{N} \cdot V) + Z$	0110
	GT	–	Dst > Src	$(N \cdot V \cdot \overline{Z}) + (\overline{N} \cdot \overline{V} \cdot \overline{Z})$	0111
	GE	XGT	Dst ≥ Src	$(N \cdot V) + (\overline{N} \cdot \overline{V})$	0101
	EQ (Z)	–	Dst = Src	$\underline{Z}$	1010
	NE (NZ)	–	Dst ≠ Src	$\overline{Z}$	1011
Compare to Zero	Z	YZ	Result = zero	$\underline{Z}$	0101
	NZ	YNZ	Result ≠ zero	$\overline{Z}$	1011
	P	–	Result is positive	$\overline{N} \cdot \overline{Z}$	0001
	N	XZ	Result is negative	$\underline{N}$	1110
	NN	XNZ	Result is nonnegative	$\overline{N}$	1111
General Arithmetic	Z	YZ	Result is zero	$\underline{Z}$	1010
	NZ	YNZ	Result is nonzero	$\overline{Z}$	1011
	C	YN	Carry set on result	$\underline{C}$	1000
	NC	YNC	No carry on result	$\overline{C}$	1001
	B (C)	–	Borrow set on result	$\underline{C}$	1000
	NB (NC)	–	No borrow on result	$\overline{C}$	1001
	V†	XN	Overflow on result	$\underline{V}$	1100
	NV†	XNN	No overflow on result	$\overline{V}$	1101

Note: A mnemonic code in parentheses is an alternate code for the preceding code.
† Also used for window clipping
\+ Logical OR
· Logical AND
¯ Logical NOT

6.8 Shift Instructions

The TMS34010 supports right-shift, left-shift, and circular-shift (rotate) instructions. These instructions use register-direct operands or a combination of register-direct and immediate operands.

Table 6-12. Summary of Shift Instructions

Instruction	Description	Instruction	Description
RL *Rs, Rd* **RL** *K, Rd*	Rotate left	**SRA** *Rs, Rd* **SRA** *K, Rd*	Shift right arithmetic
SLA *Rs, Rd* **SLA** *K, Rd*	Shift left arithmetic	**SRL** *Rs, Rd* **SRL** *K, Rd*	Shift right logical
SLL *Rs, Rd* **SLL** *K, Rd*	Shift left logical		

These instructions left-rotate, left-shift, or right-shift the contents of the destination register by the value of a 5-bit constant **or** by the value specified in the 5 LSBs of the source register. (Note that the **SRA** *Rs, Rd* and **SRL** *Rs, Rd* use the 2s complement value of the 5 LSBs in Rs.)

- The **RL** instruction left-rotates the contents of the destination register by. The bits shifted out of the MSB are shifted into the LSB. The C (carry) bit is set to the final value shifted out of the MSB.

- The **SLA** instruction left shifts the contents of the destination register. 0s are shifted into the LSBs. The MSBs are shifted out through the C (carry) bit so that the C bit is set to the final value shifted out of the MSB. If either the N (sign) bit or any of the bits shifted out differ from the original sign bit, the V (overflow) bit is set.

- The **SLL** instruction left shifts the contents of the destination register. 0s are shifted into the LSBs. The MSBs are shifted out through the C (carry) bit so that the C bit is set to the final value shifted out of the MSB. The main difference between SLL and SLA is that SLL does not check to see if the sign bit changes.

- The **SRA** instruction right shifts the contents of the destination register. The value of the sign bit is shifted into the MSBs; this sign-extends the value and preserves the original value of the sign bit. The LSBs are shifted out through the C (carry) bit so that the C bit is set to the final value shifted out of the LSB.

- The **SRL** instruction right shifts the contents of the destination register. 0s are shifted into the MSBs, beginning with bit 31. The LSBs are shifted out through the C (carry) bit so that the C bit is set to the final value shifted out of the LSB. The main difference between SRL and SRA is that SRL does not preserve the original value of the sign bit.

6.9 XY Instructions

The TMS34010 allows you to use XY addresses. This is useful for specifying pixel addresses on the screen. Many of the graphics instructions use XY addressing; the TMS34010 instruction set also supports several other instructions that allow you to manipulate XY addresses.

An XY address is a 32-bit address that is divided into two parts. The 16 LSBs of the address are the X half of the address or register; the 16 MSBs of the address are the Y half of the address or register. The two parts are treated as completely separate values; for example, using the ADDXY instruction, the X half does not propagate into the Y half.

Table 6-13 summarizes the instructions that use XY addresses.

Table 6-13. Summary of XY Instructions

Instruction	Description	Instruction	Description
ADDXY *Rs, Rd*	Add registers in XY	**PIXBLT B, XY**	Pixel block transfer (binary to XY)
CPW *Rs, Rd*	Compare point to window	**PIXBLT L, XY**	Pixel block transfer (linear to XY)
CMPXY *Rs, Rd*	Compare registers in XY mode	**PIXBLT XY, L**	Pixel block transfer (XY to linear)
CVXYL *Rs, Rd*	Convert XY address to linear address	**PIXBLT XY, XY**	Pixel block transfer (XY to XY)
DRAV *Rs, Rd*	Draw and advance	**PIXT** *Rs, *Rd*.**XY**	Pixel transfer (register to indirect XY)
FILL XY	Fill array with processed pixels	**PIXT** **Rs*.**XY**, *Rd*	Pixel transfer (indirect XY to register)
LINE [**0, 1**]	Line draw with XY addressing	**PIXT** **Rs*.**XY**, **Rd*.**XY**	Pixel transfer (indirect XY to indirect XY)
MOVX *Rs, Rd*	Move X half of Rs to X half of Rd	**SUBXY** *Rs, Rd*	Subtract registers in XY mode
MOVY *Rs, Rd*	Move Y half of Rs to Y half of Rd		

- The **PIXBLT** and **FILL** instructions in Table 6-13 use XY source and/or destination addresses.
- The **PIXT** instructions in Table 6-13 use the contents of registers as XY addresses.
- The **LINE** instruction draws a line along points that are calculated as XY addresses.
- The move instructions in Table 6-13 (**MOVX** and **MOVY**) move the X or Y half of a source register into the X or Y half of a destination register.
- The arithmetic and logical instructions in Table 6-13 (**ADDXY**, **SUBXY**, and **CMPXY**) add, subtract, or compare the X and Y halves of the registers separately.

Section 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 .copy 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) contains 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:

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. The *macro name* is the name of the macro. A macro name may be 1 to 32 alphanumeric characters; it must begin with a letter. The *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.
$IF	***Begin Conditional Block Directive*** **Syntax:** **$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 can 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, and the macro filename's extension must be .asm. The files in a macro library must be unassembled source files. 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`:

```
gspar -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 **$MACRO** *[parm$_1$] [,parm$_2$] ... [,parm$_n$]*

.
.
.

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 can 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 parameters.

$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 can 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 can 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. Figure 7-1 shows an example of a macro definition, how it could be called, and how it would be expanded in the source code.

Macro Definition: The following code defines a macro, **swap**, that has two parameters.

```
0001                    ;-----------------------------------------------
0002                    ;|                *** swap ***
0003                    ;-----------------------------------------------
0004                    swap  $MACRO  R1,R2       ; begin macro definition
0005                          xor     :R1:,:R2:   ; model statement
0006                          xor     :R2:,:R1:   ; model statement
0007                          xor     :R1:,:R2:   ; model statement
0008                          $END                ; end macro definition
```

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

```
0009                    ;-----------------------------------------------
0010                    ;|              *** Macro call ***
0011                    ;-----------------------------------------------
0012 00000000                 swap    A0,A1       ; macro call
```

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.

```
0001 00000000    5601         xor     A0,A1
0002 00000010    5620         xor     A1,A0
0003 00000020    5601         xor     A0,A1
```

Figure 7-1. An Example of a Macro Definition, Call, and Expansion

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 Parameters

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

The assembler assigns initial values to macro parameters when the macro is called. For example, consider the following macro definition line:

```
ADDUP   $MACRO    val1,val2,sum
```

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

The value that is assigned to a macro parameter is called a **string value**. The assembler will substitute a parameter's string value into a model statement when you enclose the parameter name in colons. Parameters can be used this way anywhere in a model statement (as a label, an operand, etc.).

Figure 7-2 shows a macro that has two parameters.

```
0001                            pac_4X8  $MACRO   pvalue
0002                            * Make sure these are in the same word
0003                                     .even
0004                                     $LOOP    8
0005                                     .field   :pvalue:, 4
0006                                     $ENDLOOP
0007                                     $END
0008
0009                    0008    pixval   .set     08h
0010
0011 00000000                             pac_4X8 pixval
0001                            * Make sure these are in the same word
0002 00000000                            .even
0003 00000000           0008             .field   pixval, 4
0004 00000004           0008             .field   pixval, 4
0005 00000008           0008             .field   pixval, 4
0006 0000000C           0008             .field   pixval, 4
0007 00000010           0008             .field   pixval, 4
0008 00000014           0008             .field   pixval, 4
0009 00000018           0008             .field   pixval, 4
0010 0000001C           0008             .field   pixval, 4
```

Figure 7-2. An Example of Using Parameter Values

The `pac_4X8` macro packs 8 4-bit pixels into 32 bits. The parameter `pvalue` is assigned a value that corresponds to the value that is passed when the macro is called.

7.5 Conditional Blocks

The $IF, $ELSE, and $ENDIF directives are used to construct conditional blocks within macro definitions. Conditional blocks can 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 *well-defined expression*

code to assemble if expression is true (≠ 0)

$ELSE

code to assemble if expression is false (= 0)

$ENDIF

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.

Figure 7-3 shows an example of a macro with a conditional block.

```
0001                        CMPR    $MACRO  p1,p2
0002                                $IF     :p1: <> :p2:
0003                                .string "not equal"
0004                                $ELSE
0005                                .string "equal"
0006                                $ENDIF
0007                                $ENDM
0008
0009                  0001  sym1    .set    1
0010                  0002  sym2    .set    2
0011
0012 00000000                       CMPR    sym1, sym2
0001 00000000           6E          .string "not equal"
     00000008           6F
     00000010           74
     00000018           20
     00000020           65
     00000028           71
     00000030           75
     00000038           61
     00000040           6C
```

Figure 7-3. An Example of a Conditional Block

7.6 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 can be any legal expression or macro expression.

The same restrictions apply to the declaration of a repeatable block as apply to conditional blocks. You can nest up to 10 blocks; you can nest conditional blocks within repeatable blocks, and repeatable blocks within conditional blocks. The assembler checks to see if blocks are nested properly; if they are not, the assembler produces an error message. 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.

Figure 7-4 shows an example of a repeatable block.

```
0001                          fill_ar  $MACRO   fill_val
0002                                   MOVI     :fill_val:, A0
0003                                   $LOOP    4
0004                                   MOVE     A0, *A1+
0005                                   $ENDLOOP
0006                                   $END
0007
0008 00000000                 array1   .bss     W_ar_4, 4*32
0009 00000000      05A1                MOVE     @array1, A1
     00000010 00000000+
0010 00000030                          fill_ar  32h
0001 00000030      09C0                MOVI     32h, A0
     00000040      0032
0002 00000050      9001                MOVE     A0, *A1+
0003 00000060      9001                MOVE     A0, *A1+
0004 00000070      9001                MOVE     A0, *A1+
0005 00000080      9001                MOVE     A0, *A1+
```

Figure 7-4. An Example of a Repeatable Block

7.7 Unique Labels

Labels must be unique. If you use an ordinary label in a macro, and the macro is expanded more than once, the label in the macro defines the label/symbol more than once – *this is illegal*. The macro language supports a special form of label that allows you to create unique labels within macros. To form a unique label, *simply follow the label name with a question mark*; the syntax for a unique label is:

label?

Symbols that are defined in this manner can be used like any other symbol; you can declare them as global symbols, you can use them in expressions, etc.

Figure 7-5 shows an example of a macro with unique labels.

```
0001                          copyx   $MACRO    A,B,length
0002                          loop?
0003                                  TRAP      29
0004                                  MOVE      :A:, *:B:+, 1
0005                                  DSJ       :length:, loop?
0006                                  $ENDM
0007
0008 00000090      09C1               MOVI      32, A1
     000000A0      0020
0009 000000B0      09D3               MOVI      4, B3
     000000C0      0004
0010 000000D0      09D4               MOVI      0FFFh, B4
     000000E0      0FFF
0011 000000F0                         copyx     B3,B4,A1
0001 000000F0                 loop?
0002 000000F0      091D               TRAP      29
0003 00000100      9274               MOVE      B3, *B4+, 1
0004 00000110      3C61               DSJ       A1, loop?
0012
0013 00000000      0770               SETF      16, 1, 1
0014 00000010      09D1               MOVI      28, B1
     00000020      001C
0015 00000030      09C1               MOVI      32, A1
     00000040      0020
0016 00000050      09E2               MOVI      0C0000000h, A2
     00000060 C0000000
0017 00000080      091D               TRAP      29
0018 00000090                         copyx     A1,A2,B1
0001 00000090                 loop?
0002 00000090      091D               TRAP      29
0003 000000A0      9222               MOVE      A1, *A2+, 1
0004 000000B0      3C71               DSJ       B1, loop?
```

Figure 7-5. An Example of Unique Labels

Section 8

Archiver Description

The TMS34010 archiver lets you combine several individual files into a single file called an **archive** or a **library**. Each file within the archive is called a **member**. Once you have created an archive, you can use the archiver to add more files to the library, delete or replace existing members, or extract members.

You can build libraries out of any type of files. Both the assembler and the linker accept archive libraries as input; the assembler can use libraries that contain individual source files, and the linker can use libraries that contain individual object files.

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

You can also use the archiver to build macro libraries. You can create several separate source files, each of which contains a single macro, and then use the archiver to collect these macros into a single, functional group. The .mlib assembler directive lets you specify the name of a macro library to the assembler; during the assembly process, the assembler will search the specified library for the macros that you call. Section 7 discusses macros and macro libraries in detail.

This section contains the following topics:

8.1 Archiver Development Flow

Figure 8-1 shows the archiver's role in the assembly language development process. Both the assembler and the linker accept libraries as input.

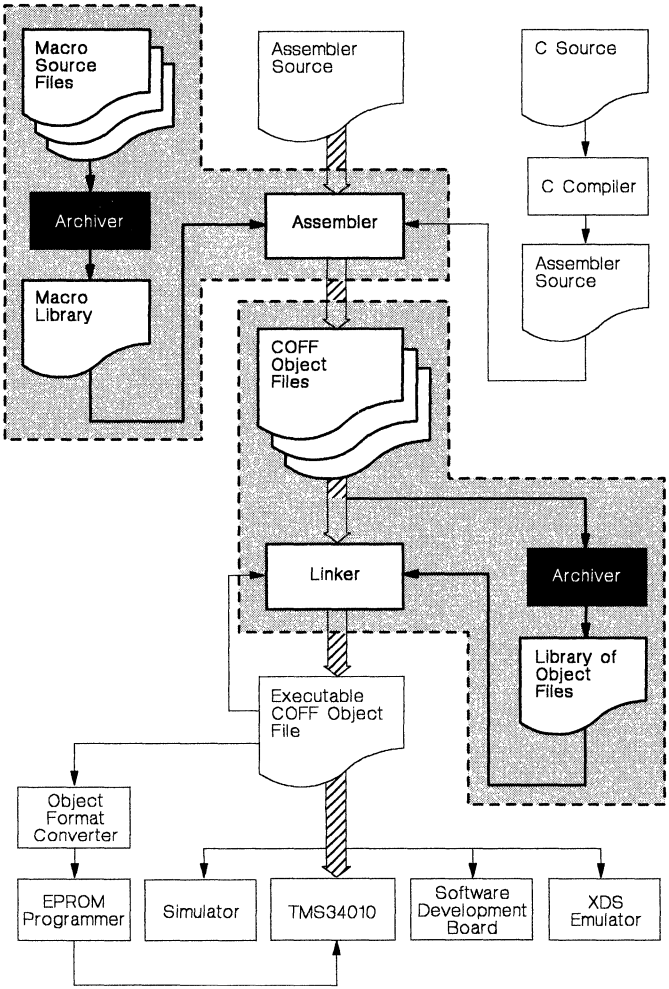


Figure 8-1. Archiver Development Flow

8.2 Invoking the Archiver

To invoke the archiver, enter:

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

gspar is the command that invokes the archiver; *libname* names an archive library. If you don't specify an extension for *libname*, the archiver uses the default extension ***.lib***. The *filenames* name individual member files that are associated with the library. If you don't specify an extension for a *filename*, the archiver uses the default extension ***.obj***.

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

a adds the specified files to the library. Note that this command **does not replace** an existing member that has the same name as an added file; it simply *appends* new members to the end of the archive. It is possible to have several members with the same name in an archive. If you want to *replace* existing members, use the **r** command.

d deletes the specified members from the library. If the library contains more than one member with a specified name, the archiver deletes the first member that has that name.

r replaces the specified members in the library. If the library contains more than one member with a specified name, the archiver replaces the first member that has that name. If you don't specify any filenames, the archiver replaces the library members with files of the same name in the current directory. If the specified file is not found in the library, the archiver adds it instead of replacing it.

t prints a table of contents of the library. If you specify filenames, only those files are listed. If you don't specify any filenames, the archiver lists all the members in the specified library.

x extracts the specified files. If you don't specify any member names, the archiver extracts all the members in the library. When the archiver extracts a member, it simply copies the member into the current directory; it *doesn't* remove it from the library.

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

e tells the archiver not to use the default extension .obj for member names.

q (quiet) suppresses the banner and status messages.

s prints a list of the global symbols that are defined in the library. (This option is valid only with the -a, -r, and -d commands.)

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

8.3 Archiver Examples

Here are some examples of using the archiver.

- **Example 1:**

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

```
gspar -a function sine cos flt
GSP Archiver            Version x.xx 87.160
(c) Copyright 1987, Texas Instruments Inc.
  ==>  new archive 'function.lib'
  ==>  building archive 'function.lib'
```

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

- **Example 2:**

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

```
gspar -t function
GSP Archiver            Version x.xx 87.160
(c) Copyright 1987, Texas Instruments Inc.
         FILE NAME    SIZE   DATE
------------------   -----   ------------------------
          sine.obj     248   Mon Nov 19 01:25:44 1984
           cos.obj     248   Mon Nov 19 01:25:44 1984
           flt.obj     248   Mon Nov 19 01:25:44 1984
```

- **Example 3:**

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

```
gspar -a function.fn sine.asm cos.asm flt.asm
GSP Archiver            Version x.xx 87.160
(c) Copyright 1987, Texas Instruments Inc.
  ==>  add 'sine.asm'
  ==>  add 'cos.asm'
  ==>  add 'flt.asm'
  ==>  building archive 'function.fn'
```

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

- **Example 4:**

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

```
gspar -as function tan.obj arctan.obj area.obj
GSP Archiver            Version x.xx 87.160
(c) Copyright 1987, Texas Instruments Inc.
  ==>  symbol defined: 'K2'
  ==>  symbol defined: 'Rossignol'
  ==>  building archive 'function.lib'
```

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

- **Example 5:**

  If you want to modify a member in a library, you can extract it, edit it, and replace it. In this example, assume there's a library named `macros.lib` that contains the members `push.asm`, `pop.asm`, and `swap.asm`.

  ```
  gspar -x macros push.asm
  ```

  The archiver makes a copy of `push.asm` and places it in the current directory; it doesn't remove `push.asm` from the library, though. Now you can edit the extracted file. To replace the copy of `push.asm` that's in the library with the edited copy, enter:

  ```
  gspar -r macros push.asm
  ```

Section 9

Linker Description

The TMS34010 linker creates executable modules by combining COFF object files. The concept of COFF *sections* is basic to linker operation; Section 3 discusses COFF in detail.

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, output section definition, and address binding. The language 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:

9.1 Linker Development Flow

Figure 9-1 illustrates the linker's role in the assembly language development process. The linker accepts several types of files as input, including object files, command files, libraries, and partially linked files. The linker creates an executable COFF object module that can be downloaded to one of several development tools or executed by a TMS34010.

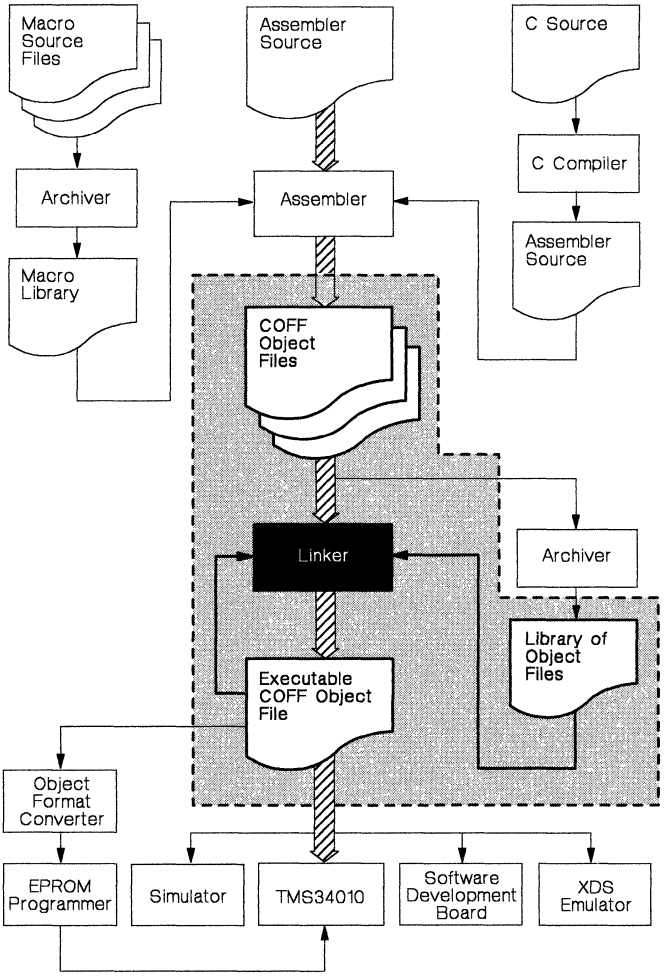


Figure 9-1. Linker Development Flow

9.2 Invoking the Linker

The general syntax for invoking the linker is:

gsplnk *[-options] filename$_1$ ... filename$_n$*

gsplnk is the command that invokes the linker. The *options* (discussed in Section 9.3) can appear anywhere on the command line or in a linker command file. The *filenames* can be object files, linker command files, or archive libraries. (The linker can determine whether the input file is an object file or an ASCII file that contains linker commands.) The default extension for all files is ***.obj***; any other extension must be explicitly specified. The default filename for the executable output module 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`.

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

- Enter the **gsplnk** 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 creates an object file with the default name `a.out`.

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

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

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

Now you can invoke the linker from the command line; specify the command file name as an input file: **`gsplnk linker.cmd`**.

When you use a command file, you can enter additional options and files on the command line. For example, you could enter **`gsplnk -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`, `file2`, and `file3`. This example creates an output file called `link.out` and a map file called `link.map`.

9.3 Linker Options

Linker options control linking operations. Options can be placed on the command line or in a command file. Linker options must be preceded by a hyphen (-). The order in which options are specified is unimportant, except for the -l and -i options. Options are separated from arguments (if they have them) by an optional space. 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 is specified.
-ar	Produce a relocatable, executable object module.
-c	Use linking conventions defined by the ROM autoinitialization model of the C compiler.
-cr	Use linking conventions defined by the RAM autoinitialization model of the C compiler.
-e *global symbol*	Define a *global symbol* that specifies the primary entry point for the output module.
-f *fill value*	Set the default fill value for holes within output sections; *fill value* is a 4-byte constant.
-h	Make all global symbols static.
-i *dir*	Alter the library-search algorithm to look in *dir* before looking in the default location. This option must appear before the -l option.
-l *filename*†	Name an archive library as the linker input; *filename* is an archive library name.
-m *filename*†	Produce a map or listing of the input and output sections, including holes, and place the listing in *filename*.
-o *filename*†	Name the executable output module. The default filename is *a.out*.
-q	Request a quiet run (suppress the banner).
-r	Retain relocation entries in the output module.
-s	Strip symbol table information and line number entries from the output module.
-u *symbol*	Place an unresolved external symbol into the output module's symbol table.

† The *filename* must follow operating system conventions

9.3.1 Relocation Capability (-a and -r Options)

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

- Producing an Absolute Output Module (-a Option)

  When you use the **-a** option without the -r option, the linker produces an *absolute, executable* output module. Absolute modules contain *no* relocation information. Executable files:

  - Contain special symbols defined by the linker (Section 9.11.4, page 9-31, describes these symbols),
  - Contain an optional header that describes information such as the program entry point, **and**
  - Contain *no* unresolved references.

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

```
gsplnk -a file1.obj file2.obj
```

> **Note:**
>
> If you do not use the -a or the -r option, the linker acts as if you specified -a.

- Producing a Relocatable Output Module (-r Option)

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

  The linker produces an *unexecutable* file when you use the -r option without -a. A file that is not executable does not contain special linker symbols or an optional header. The file may contain unresolved references, but these references do not prevent creation of an output module.

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

  ```
  gsplnk -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, Section 9.13, page 9-37.)

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

  If you invoke the linker with both the -a and the -r options, the linker produces an *executable, relocatable* object module. The output file contains the special linker symbols, contains an optional file header, and all symbol references are resolved; however, the relocation information is retained.

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

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

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

- Relocating or Relinking an Absolute Output Module

  The linker issues a warning message (but continues executing) when it encounters a file that contains no relocation or symbol table information. Relinking an absolute file can only be successful if each input file contains no information that needs to be relocated (that is, if each file has no unresolved references and is bound to the same virtual address that it was bound to when the linker created it).

9.3.2 C Language Options (-c and -cr Options)

The -c and -cr options cause the linker to use linking conventions that are required by the TMS34010 C compiler.

- The **-c** option tells the linker to use the ROM autoinitialization model.
- The **-cr** option tells the linker to use the RAM autoinitialization model.

For more information about linking C code, see section Section 9.14 on page 9-38.

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

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

The linker assigns one of four possible values to the entry point. These values are listed below in the order in which the linker tries to use them. If you use one of the first three values, it must be an external symbol in the symbol table. Possible entry point values include:

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.

2) The value of symbol `_c_int00` (if present). `_c_int00` **must** be the entry point if you are linking code produced by the C compiler.

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

4) Zero (default value).

This example links `file1.obj` and `file2.obj` The symbol `begin` is the entry point; `begin` must be defined as external in `file1` or `file2`.

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

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

The -f option fills the holes formed within output sections or initializes uninitialized sections when they are combined with initialized sections. This allows you to initialize memory areas during link time without reassembling a source file. The argument *cc* is a 4-byte constant (up to eight hexadecimal digits). When -f is not used, the default fill value is 0.

This example fills holes with the hexadecimal value $AABBCCDD_{16}$:

```
gsplnk -f 0AABBCCDDh file1.obj file2.obj
```

9.3.5 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 are defined, so no external references are possible. For example, assume `file1.obj` and `file2.obj` both define global symbols called `EXT`. By using the -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`.

```
gsplnk -h file1.obj file2.obj
```

9.3.6 Alter the Library Search Algorithm (-*idir* Option/C_DIR)

Usually when you want to specify a library as linker input, you simply enter the library name as you would any other input filename; the linker looks for the library in the current directory. For example, suppose the current directory contains the library `object.lib`. Assume that this library defines symbols that are referenced in the file `file1.obj`. This is how you link the files:

```
gsplnk file1.obj object.lib
```

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

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

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

9.3.6.1 -i Linker Option

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

As an example, assume that two archive libraries called `r.lib` and `lib2.lib` reside in directories called:

- `\ld` and `\ld2`, respectively (PC/MS-DOS),
- `[ld]` and `[ld2]`, respectively (VAX/VMS), **or**
- `/ld` and `/ld2`, respectively (VAX/UNIX and System V).

You can use both libraries during a link:

DOS: `gsplnk f1.obj f2.obj -i\ld -i\ld2 -lr.lib -llib2.lib`

VMS: `gsplnk f1.obj f2.obj -i[ld] -i[ld2] -lr.lib -llib2.lib`

UNIX: `gsplnk f1.obj f2.obj -i/ld -i/ld2 -lr.lib -llib2.lib`

9.3.6.2 Environment Variable (C_DIR)

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

DOS: **set C_DIR**=*pathname;another pathname ...*

VMS: **assign C_DIR** *"pathname;another pathname ... "*

UNIX: **setenv C_DIR** *"pathname;another pathname ... "*

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

As an example, assume that two archive libraries called `r.lib` and `lib2.lib` reside in directories called:

- `\ld` and `\ld2`, respectively (PC/MS-DOS),
- `[ld]` and `[ld2]`, respectively (VAX/VMS), **or**
- `/ld` and `/ld2`, respectively (VAX/UNIX and System V).

You can use both libraries during a link; set the environment variable first:

DOS:
```
set    C_DIR=\ldir;\ldir2
gsplnk f1.obj f2.obj -lr.lib -llib2.lib
```

VMS:
```
assign C_DIR "/ldir;ldir2"
gsplnk f1.obj f2.obj -lr.lib -llib2.lib
```

UNIX:
```
setenv C_DIR "/ldir;ldir2"
gsplnk f1.obj f2.obj -lr.lib -llib2.lib
```

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

DOS: `set C_DIR=`

VMS: `deassign C_DIR`

UNIX: `setenv C_DIR " "`

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

Section 9.5 (page 9-13) contains more information about object libraries.

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

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

- Memory configuration,
- Input and output section allocation, **and**
- The addresses 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`:

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

See Section 9.15 (page 9-41) for an example of a map file.

9.3.8 Name 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 gives it the default name `a.out`. If you want to assign a different name to the output module, use the -o option. The *filename* is the new output module name.

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

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

9.3.9 Specify a Quiet Run (-q Option)

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

9.3.10 Strip Symbolic Information (-s Option)

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

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

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

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

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

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

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

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

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

9.4 Linker Command Files

Linker command files allow you to put linking information in a file; this is useful when you often invoke the linker with the same information. Linker command files are also useful because they allow you to use the MEMORY and SECTIONS directives to customize your application. You must use these directives in a command file; you cannot use them on the command line. Command files are ASCII files that contain one or more of the following:

- Input file names, which specify object files, object libraries, or other command files. (If a command file calls another command file as input, this statement must be the *last* statement in the calling command file. The linker does not return from the called command file.)
- Linker options, which can be used in the command file in the same manner that they are used on the command line.
- The MEMORY and SECTIONS linker directives. The MEMORY directive defines the target memory configuration. The SECTIONS directive controls how sections are built and allocated.
- Assignment statements, which define and assign values to global symbols.

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

gsplnk *command file name*

The linker processes input files in the order that it encounters them. If the linker recognizes a file as an object file, it links it. Otherwise, it assumes a file is a command file, and begins reading and processing commands from it.

Figure 9-2 shows a sample linker command file called `link.cmd`.

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

Figure 9-2. An Example of a Linker Command File

The sample file in Figure 9-2 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:

```
gsplnk link.cmd
```

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

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

The linker processes the command file as soon as it encounters it, so `a.obj` and `b.obj` are linked into the output module before `c.obj` and `d.obj`.

You can specify multiple command files for a single link. 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:

```
gsplnk names.lst dir.cmd
```

One command file can also call another command file; this type of nesting is limited to 16 levels. If a command file names another command file as input, this statement must be the last statement in the calling command file.

Blanks and blank lines that appear in a command file are insignificant except as delimiters. This applies to the format of linker directives in a command file, also. Figure 9-3 shows a sample command file that contains linker directives. (Linker directive formats are discussed in later sections.)

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

MEMORY                           /* MEMORY directive     */
{
  RAM:  origin =   100h ,   length = 0100h
  ROM:  origin = 01000h ,   length = 0100h
}

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

Figure 9-3. An Example of a Command File with Linker Directives

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

align	l (lowercase "L")	origin
ALIGN	len	ORIGIN
block	length	page
BLOCK	LENGTH	PAGE
COPY	MEMORY	range
DSECT	NOLOAD	SECTIONS
group	o	spare
GROUP	org	

9.5 Object Libraries

An archive library is a partitioned file that contains complete object files as members. Usually, a group of related modules are collected together into a library. When you specify an object library as linker input, the linker links in any members that define existing unresolved symbol references. You can use the TMS34010 archiver to build and maintain libraries (see Section 8).

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

The order in which you specify libraries as input is important because the linker includes only those members that resolve symbols that are undefined when the library is searched. The same library can be specified as often as necessary; it is searched each time it is included. A library contains a table that lists all external symbols defined in the library; the linker searches through the table until it determines it cannot use the library to resolve any more references.

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

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

If you enter: `gsplnk 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`.
- Both `liba.lib` and `libc.lib` are in the current directory.

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

then the references to `clrscr` are satisfied by member 1 of `libc.lib`. If none of the linked files reference symbols defined in a library, you can use the -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:

```
gsplnk -u rout1 libc.lib
```

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

Section 9.3.6 (page 9-7) describes methods for specifying directories that contain object libraries.

9.6 The MEMORY Directive

The linker determines where output sections should be allocated into memory; the linker must have a model of target memory to accomplish this task. The MEMORY directive allows you to specify a model of target memory, so you can define the types of memory your system contains and the address ranges they occupy. The linker maintains the model as it allocates output sections, and uses the model to determine which locations in the target system can be used for the linked program.

The memory configurations of TMS34010 systems differ from application to application. The MEMORY directive allows you to specify a variety of configurations. After you use the MEMORY directive to define a memory model, you can use the SECTIONS directive to allocate output sections into defined memory.

9.6.1 Default Memory Model

If you do not use the MEMORY directive, the linker uses a default memory model which assumes that the full 32-bit address space (2^{32} locations) is present in the system and available for use. For more information about this default model, see Section 9.9 on page 9-26.

9.6.2 MEMORY Directive Syntax

The MEMORY directive identifies ranges of memory that are physically present in the target system and can be used by a program. Each memory range has a *name*, a *starting address*, and a *length*.

When you use the MEMORY directive, be sure to identify **all** the memory ranges that are available to load object code into. Memory that is defined by the MEMORY directive is **configured memory**; any memory that you do not explicitly account for with the MEMORY directive is **unconfigured memory**. (Note that under the default memory model, the entire address space is configured.) The linker can only use configured memory; it cannot place any part of a program into unconfigured memory.

The MEMORY directive is specified in a command file by the word MEMORY (uppercase), followed by a list of memory range specifications enclosed in braces. The MEMORY directive in Figure 9-4 defines a system that has 4K of ROM at address 0 and 8K of RAM at address 0E000h.

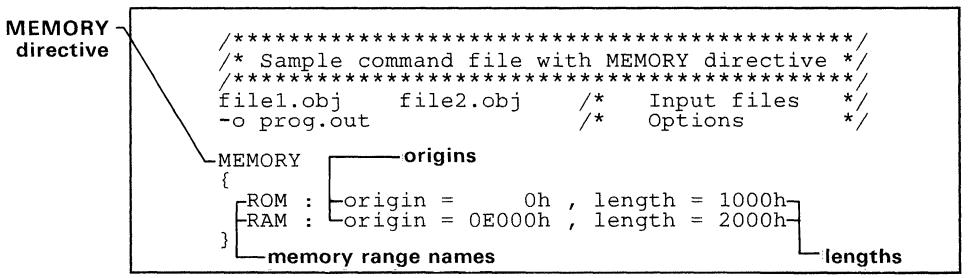


Figure 9-4. An Example of the MEMORY Directive

The general syntax for the MEMORY directive is:

```
MEMORY
{
    name 1 [(attr)]  : origin = constant , length = constant
    .
    .
    name 1 [(attr)]  : origin = constant , length = constant
}
```

(Boldfaced items should 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 significance to the program; they simply identify memory ranges for the linker. 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 optional attributes associated with the named range. Valid attributes include: **R** (readable memory), **W** (writable memory), **X** (executable memory), and **I** (initializable memory). The attribute list must be enclosed in parentheses. If you do not specify any attributes for a memory range, then the range *has all four attributes*. All memory in the default model has all four attributes. The following example specifies a memory range with the R and X attributes:

```
MEMORY
{    ROM (RX) : origin = 0 , length = 01000h}
```

origin specifies the starting address of a memory range. It may be entered as *origin*, *org*, or *o*. The value, specified in bits, is a long integer constant, and may be decimal, octal, or hexadecimal. The comma that follows the origin specification is optional.

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

Figure 9-5 illustrates the memory map defined by Figure 9-4.

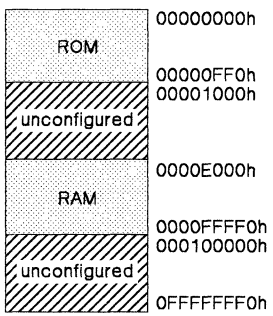


Figure 9-5. Memory Map Defined in Figure 9-4

9.7 The SECTIONS Directive

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

- Describes how input sections 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.

9.7.1 Default Sections Configuration

If you do not specify a SECTIONS directive, the linker uses a default algorithm for combining and allocating the sections. Section 9.9 (page 9-26) describes this algorithm in detail.

9.7.2 SECTIONS Directive Syntax

The SECTIONS directive is specified in a command file by the word SECTIONS (uppercase), followed by a list of output section specifications enclosed in braces. Figure 9-6 contains an example of the SECTIONS directive.

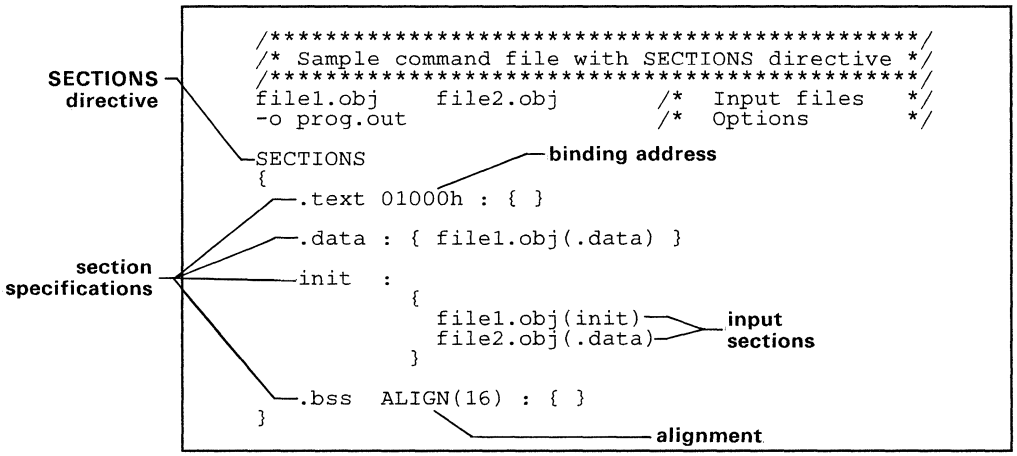


Figure 9-6. An Example of the SECTIONS Directive

The general syntax for 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 a section in the output file.) The syntax for a section specification is:

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

(Boldfaced portions should be entered as shown.)

name names the section in the output file. Only the first 8 characters of output section names are significant.

binding is optional, and assigns the section to a specific physical address in the target memory. Section 9.7.4 (page 9-20) discusses assigning an address to an output section.

align(n) is optional, and specifies that the section should be aligned on an n-bit boundary (the actual address is determined by the linker). Section 9.7.4 (page 9-20) 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; if the braces are empty (no input sections are specified), the linker includes all input sections with the same name as the output section. Section 9.7.3 (page 9-18) 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. Section 9.11 (page 9-29) discusses linker assignment statements, and Section 9.12 (page 9-32) contains more information about holes.

fill value is optional, and specifies a value for filling holes in the section. See Section 9.12 (page 9-32) 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 9.7.4 (page 9-20) discusses named memory.

Figure 9-7 shows how the sections in Figure 9-6 (page 9-16) are allocated. Figure 9-6 defines four output sections, .text, .data, init, and .bss:

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

  An address was specified for the .text section (this is called *binding*), so the .text output section will begin at address 01000h 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-bit boundary.

Figure 9-7 shows how the sections in Figure 9-6 are allocated.

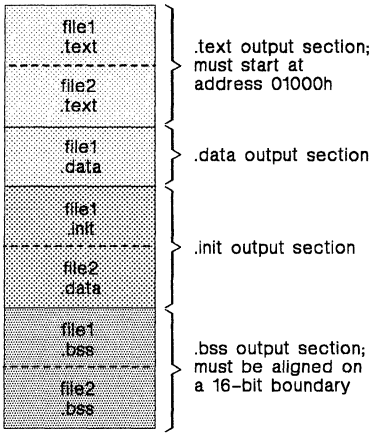


Figure 9-7. Section Allocation Defined by Figure 9-6

9.7.3 Specifying Input Sections

An input section specification identifies the sections from input files that are combined to form an output section. The linker combines input sections by concatenating them in the order 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.

Figure 9-8 shows the most common type of section specification; note that *no* input sections are listed.

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

Figure 9-8. The Most Common Method of Specifying Section Contents

In the example shown in Figure 9-8, 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 it encountered them in the input files. The

linker performs similar operations with the .data and .bss sections. You can use this type of specification for *any* output section.

You can explicitly specify the input sections that form an output section. Each input section is identified by its filename and section name:

```
SECTIONS
{
 .text :                 /* Build .text output section      */
 {
  f1.obj(.text)          /* Link .text section from f1.obj  */
  f2.obj(sec1)           /* Link sec1 section from f2.obj   */
  f3.obj                 /* Link ALL sections from f3.obj   */
  f4.obj(.text, sec2)    /* Link .text and sec2 from f4.obj */
 }
}
```

Note that it is not necessary for 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 are automatically linked in at the end of the output section. For example, if the linker found more .text sections in the preceding example, and these .text sections *were not* specified anywhere in the SECTIONS directive, then the linker would concatenate these extra sections after `f4.obj(sec2)`.

The specifications in Figure 9-8 are actually a shorthand method for the following:

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

The `*(.text)` means *the unallocated .text sections from all the input files* This format is useful when:

- You want the output section to contain all input sections that have a certain name, but the output section name is different from the input sections' name.
- You want the linker to allocate the input section *before* it processes additional input sections or commands within the braces.

Here's an example that uses this method:

```
SECTIONS
{
     .text:   {
                 abc.obj(xqt)
                 *(.text)
              }
    .data  :  {
                 *(.data)
                 fil.obj(table)
              }
}
```

In this example, the .text output section contains a named section `xqt` from file `abc.obj`, followed by *all* the .text input sections. The .data section con-

tains *all* the .data input sections, followed by a named section `table` from file `fil.obj`. Note that this method includes all the *unallocated* sections. For example, if one of the .text input sections was already included in another output section when the linker encountered `*(.text)`, the linker could not include that first .text input section in the second output section.

9.7.4 Specifying the Address of Output Sections (Allocation)

After you specify the contents of each output section, you must specify the physical location in target memory where the output section will be loaded. Each section has an address field in its section header that tells a loader where the section should go. The process of calculating the addresses of the output sections is called **allocation**.

If you do not specify an explicit starting address for an output section, the linker uses a default algorithm to allocate the section. Generally, the linker puts sections where ever they 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 supply a specific address for an output section by following the section name with the address:

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

  This example specifies that the .text section must begin at location 1000h. The binding address must be a 32-bit constant.

  Output sections can be bound anywhere in configured memory (assuming there is enough space), but they cannot overlap. If there is not enough space to bind a section to a specified address, the linker issues an error message.

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

- **Alignment**

  You can tell the linker to place an output section at an address that falls on an *n*-bit 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 bits.

  The assembler also supports a method for specifying alignment. The .align assembler directive aligns code or data on a 32-word (cache) boundary. When you use .align, the assembler sets a flag that tells the linker to align the entire section. This ensures that all the alignments within the section are correct when the section is relocated.

- **Named Memory**

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

```
MEMORY
{
    ROM (RIX) :  origin = 0h,    length = 1000h
    RAM (RWIX):  origin = 3000h, 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 0h and 0FFFh. The .data and .bss sections are allocate into the named memory range RAM. It is possible to align a section within named memory; the .data section is aligned on a 64-bit boundary within RAM.

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

```
/*************************************************/
/**  .text --> executable memory                **/
/**  .data --> readable or initializable memory  **/
/**  .bss  --> readable or writable memory       **/
/*************************************************/

SECTIONS
{
  .text: {...} > (X)
  .data: {...} > (RI)
  .bss : {...} > (RW)
}
```

  In this example, the .text section can be linked into either the ROM or RAM area, since both are 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 has the W attribute.

  You cannot control where in a named memory range a section is allocated, although the linker uses lower memory addresses first and avoids fragmentation when possible. In the preceding examples, assuming no other sections were bound to addresses that would interfere with this allocation process, the .text section would start at address 0. If a section must start on a specific address, use binding instead of named memory.

9.7.5 Grouping Output Sections Together

The SECTIONS directive has a GROUP option that forces several output sections to be allocated contiguously. For example, assume that is a section named `term_rec` contains a termination record for a table in the .data section. You can force the linker to allocate .data and `term_rec` together:

```
SECTIONS
{
  .text : { }           /* Normal output section        */
  .bss  : { }           /* Normal output section        */
  GROUP 1000h :         /* Specify a group of sections */
  {
      .data : { }       /* First section in the group   */
      term_rec : { }    /* Follows .data in memory      */
  }
}
```

You can use binding, alignment, or named memory to allocate a GROUP of output sections in the same manner that you can allocate a single output section. In the preceding example, the GROUP is bound to address 1000h. This means that .data is allocated at 1000h, and `term_rec` follows it in memory. You can also use alignment and named memory with the GROUP option.

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

9.8 Overlay Pages

Some target systems use an overlay memory configuration, where all or part of the memory space is overlayed by shadow memory. A system can map different banks of physical memory in and out of a a single address range in response to hardware select signals. In this situation, 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, or into areas that are not mapped at load time.

The linker supports this feature by providing *overlay pages*, allowing you to define a memory model with multiple address spaces. To the linker, each possible overlay configuration represents a separate address space. 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 map sections into various pages.

9.8.1 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 that has the full 32-bit range of addressable locations. 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 TMS34010. The default memory model resides entirely on PAGE 0. If a memory range is specified without a page number, the linker allocates it into 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, assume that your system can select between three 4K banks of physical memory to map into the address space from 1000h to 2000h. Although only one bank can be selected at a time, you can initialize each bank with different data. Assume that three output sections called `sect0`, `sect1`, and `sect2` must be linked into the three banks of memory. Figure 9-9 shows the MEMORY directive that defines this configuration. Figure 9-10 (page 9-24) illustrates this configuration; it shows each available block of physical memory in the system and the section that must be loaded into it.

```
/************************************************/
/* Example of MEMORY directive with overlay pages */
/************************************************/
MEMORY
{
   PAGE 0:  ROM      : origin = 0h,       length = 1000h
            RAM      : origin = 100000h,  length = 0F00000h
            OVR_MEM  : origin = 1000h,    length = 1000h
   PAGE 1:  OVR_MEM  : origin = 1000h,    length = 1000h
   PAGE 2:  OVR_MEM  : origin = 1000h,    length = 1000h
}
```

Figure 9-9. An Example of Overlay Pages

This example defines three separate address spaces. PAGE 0 is the normal TMS34010 address space. It contains the memory ranges `ROM` and `RAM`; suppose they represent all the memory in the normal address space. PAGE 0 also contains the first bank of overlay memory (`OVR_MEM`). The other two address

spaces contain only the additional banks of overlay memory, both labeled OVR_MEM. Note that all three OVR_MEM ranges cover the same address range. This is possible because each range is on a different page, and therefore represents a different memory space.

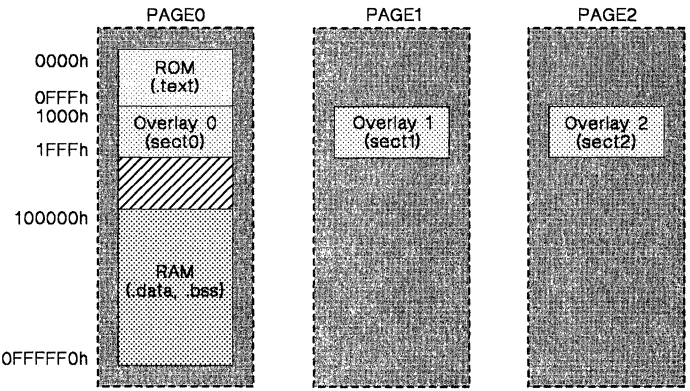


Figure 9-10. Overlay Pages Defined by Figure 9-9

9.8.2 Using Overlay Pages with the SECTIONS Directive

The SECTIONS directive tells the linker which page an output section should be linked into. Each output section 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. Figure 9-11 shows the SECTIONS directive for Figure 9-9.

```
SECTIONS
{
   .text: {} > ROM              /* Link .text in ROM on PAGE 0      */
   .data: {} > RAM              /* Link .data in RAM on PAGE 0      */
   .bss : {} > RAM              /* Link .bss in RAM on PAGE 0       */
   sect0: {} > OVR_MEM PAGE 0   /* Link sect0 into bank 0 (PAGE 0)  */
   sect1: {} > OVR_MEM PAGE 1   /* Link sect1 into bank 1           */
   sect2: {} > OVR_MEM PAGE 2   /* Link sect2 into bank 2           */
}
```

Figure 9-11. SECTIONS Directive Definition for Figure 9-9

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

The PAGE specifications 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 1000h, but *in different memory spaces*. When the program is loaded, a 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 `OVR_MEM`. This allows you to define the allocation of sections within a page, just as you can in a single memory space. For example:

```
sect1 1200h: {} PAGE 1
```

links `sect1` at address 1200h in page 1. If you do not specify any binding or named memory range for the section, the linker allocates the section wherever it can into the page (just as it does with a single memory space). For example, `sect2` could also be specified as:

```
sec2 : {} PAGE 2
```

Since `OVR_MEM` is the only memory on PAGE 2, it is not necessary (but acceptable) to specify `> OVR_MEM` for the section.

9.8.3 Syntax of Page Definitions

As the preceding examples show, 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 make up the page. Memory ranges are specified in the same manner as when no PAGE option is used. 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 names. 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 memory ranges listed outside the scope of a PAGE specification default to PAGE 0. Consider the following example:

```
MEMORY
{
            ROM    : org =     0h   len =  1000h
            EPROM  : org = 1000h    len =  1000h
            RAM    : org = 2000h    len = 0E000h
    PAGE 1: XROM   : org =     0h   len =  1000h
            XRAM   : org = 2000h    len = 0E000h
}
```

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

The link map listing 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.

9.9 Default Allocation

The MEMORY and SECTIONS directives provide flexible methods for building, combining, and allocating sections. However, any memory locations or sections that you choose *not* to specify must still be handled by the linker. The linker uses default algorithms to build, combine, and allocate sections, within the specifications you supply.

9.9.1 Allocation Algorithm

If you do not use the SECTIONS directive, the linker uses the following SECTIONS definition:

```
SECTIONS
{
        .text   ALIGN (16): { }
        .data   ALIGN (16): { }
        .bss    ALIGN (16): { }
        .cinit  ALIGN (16): { }  /* For -c and -cr */
}
```

All .text input sections are concatenated to form a .text output section in the executable output file, all .data input sections are combined to form a .data output section, and all .bss sections are combined to form a .bss output section. Each output sections is aligned on a 16-bit (word) boundary and then allocated into configured memory.

If you do not use the MEMORY directive, the linker assumes that the full 32-bit address space is configured and available. Thus, under the default algorithm, the linker allocates output sections into memory beginning at location 0.

If the input files contain named sections, the linker combines sections with the same names and allocates them following the .bss section.

Note that if you use the SECTIONS directive, the linker performs **no part** of the default allocation. Allocation is performed according to the rules specified by the SECTIONS directive and the general algorithm described in Section 9.9.2.

9.9.2 General Rules for Output Sections

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

Rule 1: As the result of a SECTIONS directive definition.

Rule 2: By combining input sections with the same names into output sections that are not defined in a SECTIONS directive.

If an output section is formed as a result of a SECTIONS directive (rule 1), its specification in the directive completely determines its contents. (Section 9.7, page 9-16, tells you 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 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; if there is no MEMORY directive, the linker uses the default configuration.

The linker's allocation algorithm attempts to minimize memory fragmentation. This allows memory to be used more efficiently and increases the probability that your program will fit into memory. This is the algorithm:

1) Output sections for which you have supplied a specific binding address are placed in memory at that address.

2) Output sections that are included in a specific named memory range or that have memory attribute restrictions are allocated. Each output section is placed into the first available space within the named area, considering alignment where necessary.

3) Any remaining sections are allocated in the order in which they are defined. Sections not defined in a SECTIONS directive are allocated in the order in which the linker encountered them. Each output section is placed in to the first available memory space, considering alignment where necessary.

9.10 Special Section Types (DSECT, COPY, and NOLOAD)

You can assign three special types to output sections: DSECT, COPY, and NOLOAD. These types affect the way that a program is treated when it is linked and loaded. You can assign a type to a section following the section definition with the type (enclosed in parentheses). For example,

```
SECTIONS
{
        sec1 200000h (DSECT)  : {f1.obj}
        sec2 400000h (COPY)   : {f1.obj}
        sec3 600000h (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 200000h. The other sections can refer to any of the global symbols in `sec1`.

- A COPY section is similar to a dummy section, but its contents and associated information are written to the output module. The `.cinit` section that contains initialization tables for the C compiler has this attribute under the RAM model.

- A NOLOAD section differs from a normal output section in one respect: the section's contents, relocation information, and line number information are not placed in the output module. The linker allocates space for it, it appears in the memory map listing, etc.

9.11 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.

9.11.1 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 defined externally in your program. If it is not, the linker defines a new symbol and enters it into the symbol table. The expression must follow the rules defined in Section 9.11.3. Assignment statements **must** be terminated with a semicolon.

The linker processes assignment statements *after* it allocates all the output sections. Thus, if an expression contains a symbol, the address used for that symbol reflects the symbol's address in the executable output file.

For example, suppose a program reads data from one of two tables identified by two external symbols, `Table1` and `Table2`. The program uses the symbol `cur_tab` as the address of the current table. `cur_tab` must point to either `Table1` or `Table2`. You could accomplish this in the assembly code, but you would need to reassemble the program in order to change tables. Instead, you can use a linker assignment statement to assign `cur_tab` at link time:

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

9.11.2 Assigning the SPC to a Symbol

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

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

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

This defines the symbol `Dstart` to be the ultimate linked address of the .data section. (`Dstart` is assigned *before* .data is allocated.) The linker will relocate all references to `Dstart`.

A special type of assignment assigns a value to the "." symbol. This adjusts the location counter within an output section and creates a hole between two input sections. Any value assigned to "." to create a hole is 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 9.12 (page 9-32).

9.11.3 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*-bit boundary within an output section (*n* is a power of 2). For example, the expression:

```
. = align(16);
```

aligns the SPC within the current section on the next 16-bit boundary. Since the align operator is a function of the current SPC, 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

9.11.4 Symbols Defined by the Linker

The linker automatically defines several symbols that a program can use at run time to determine where a section is linked. 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 directive.

Values are assigned to these symbols as follows:

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

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

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

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

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

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

cinit is assigned the first address of the .cinit section (when -c or -cr is used).

9.12 Creating and Filling Holes

The linker provides you with the ability to create areas *within output sections* that have nothing linked into them. These areas are called **holes**. In special cases, uninitialized sections can also be treated as holes. This section describes how the linker handles such holes and how you can fill holes (and uninitialized sections) with a value.

9.12.1 Initialized and Uninitialized Sections

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

Rule 1: Raw data for the *entire* section **or**

Rule 2: *No* raw data.

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

By default, the .bss section and named sections created with the .usect directive contain no raw data; such sections are uninitialized. They occupy space in the memory map, but have no actual contents. Uninitialized sections typically reserve space in RAM for variables. In the object file, an uninitialized section has a normal section header and may have symbols defined in it. However, no memory image is stored in the section.

9.12.2 Creating Holes

You can create a hole in an initialized output section. A hole is created when you force the linker to leave extra space between input sections within an output section. When such a hole is created, *the linker must follow rule 1 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 holes. There is no way to fill or initialize the space between output sections.

To create a hole in an output section, you must use a special type of linker assignment statement within an output section definition. The assignment statement modifies the SPC (denoted by ".") by either adding to it, assigning a greater value to it, or aligning it on an address boundary. The operators, expressions, and syntax of assignment statements are described in Section 9.11 (page 9-29).

The following example shows how assignment statements create holes in output sections:

```
SECTIONS
{
   outsect:
   {
       file1.obj(.text)
       . += 100h;      /* Create a hole with size 100h    */
       file2.obj(.text)
       . = align(16); /* Create a hole to align the SPC */
       file3.obj
   }
}
```

The output section `outsect` is built as follows:

- The .text section from `file1.obj` is linked in.
- The linker creates a 256-bit 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-bit 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 `. = align(16)` in the preceding example. This statement effectively aligns `file3.obj` .text to start on a 16-bit boundary within `outsect`. If `outsect` is ultimately allocated to start on an address that is not aligned, then `file3` .text will not be aligned either.

Expressions that decrement "." are illegal. For example, it is invalid to use the -= operator in an assignment to ".". The most common operators used in assignments to "." are += and align.

If an output section contains all input sections of a certain type (such as .text), you can use the following statements to create a hole at the beginning or end of the output section:

```
.text:  {  . += 100h; }   /* Hole at the beginning */
.data:  {  *(.data)
           . += 100h; }   /* Hole at the end       */
```

Another way to create a hole in an output section is to combine an uninitialized section with initialized sections to form a single output section. *In this case, the linker treats the uninitialized 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, `outsect` must also contain raw data (rule 1). Therefore, the uninitialized .bss section becomes a hole.

Note that uninitialized sections only become holes when they are combined with initialized sections. If several uninitialized sections are linked together, the resulting output section is also uninitialized.

9.12.3 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 4-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 is formed by combining an uninitialized section with an initialized section, you can specify a fill value for the uninitialized section. Follow the section name with an = symbol and a 4-byte constant:

```
SECTIONS
{
     outsect:
     {
         file1.obj(.text)
         file2.obj(.bss) = 0FFh  /* Fill this hole */
     }                                                    /*
with 000000FFh */
}
```

2) You can also specify a fill value for all the holes in an output section by supplying the fill value after the section definition:

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

3) If you do not specify an initialization for a hole, the linker fills the hole with the value specified with -f. For example, suppose the command file `link.cmd` contains the following SECTIONS directive:

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

Now invoke the linker with the -f option:

```
gsplnk -f 0FFFFFFFFh link.cmd
```

This fills the hole with 0FFFFFFFFh.

4) If you do not invoke the linker with -f, the linker fills holes with 0s.

Whenever a hole is created and filled in an initialized output section, the hole is identified in the link map along with the value the linker uses to fill it.

9.12.4 Explicit Initialization of Uninitialized Sections

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

However, you can force the linker to initialize an uninitialized section by supplying 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: {} = 11223344h  /* Fills .bss with 11223344h */
}
```

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 uninitialized sections or holes.

9.13 Partial Linking

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

Follow these guidelines for producing a file that you will relink:

- Intermediate files **must** have relocation information. Use the -r option when you link the file the first time.
- Intermediate files **must** have symbolic information. By default, the linker retains symbolic information in its output. Do not use the -s option if you plan to relink a file, because -s strips symbolic information from the output module.
- Intermediate link steps should 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 step.

The following example shows how you can use partial linking:

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

```
gsplnk -r -o tempoutl filel.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 the output file `tempout2.out`

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

`file2.com` contains:

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

- **Step 3:** Link `tempout1.out` and `tempout2.out`:

```
gsplnk -m final.map -o final.out tempoutl.out tempout2.out
```

9.14 Linking C Code

The TMS34010 C compiler produces assembly language source code that can be assembled and linked. For example, a C program consisting of modules `prog1`, `prog2`, etc., can be linked to produce an executable file `prog.out` by invoking the linker with the following command:

```
gsplnk -c -o prog.out prog1.obj prog2.obj rts.lib [flib.lib]
```

The -c option tells the linker to use special conventions that are defined by the C environment. The archive libraries `rts.lib` and `flib.lib` contain C runtime support functions.

9.14.1 Runtime Initialization

All C programs must be linked with an object module called `boot.obj`. When a program begins running, it executes `boot.obj` first. `boot.obj` contains code and data for initializing the runtime environment; the module performs the following tasks:

- It sets up the system stack.
- It processes the runtime initialization table and autoinitializes global variables (in the ROM model).
- It disables interrupts and calls `_main`.

The runtime support object library, `rts.lib`, contains `boot.obj`. You can:

- Use the archiver to extract `boot.obj` from the library and then link the module in directly, **or**
- Include `rts.lib` as an input file (the linker automatically extracts `boot.obj` when you use the -c or -cr option.)

9.14.2 Object Libraries and Runtime Support

The *TMS34010 C Compiler Reference Guide* describes additional runtime support functions that are included in `rts.lib`. If your program uses any of these functions, you must link `rts.lib` with your object files.

If you are using floating point arithmetic, you must include the TMS34010 floating-point library in your link. This library, called `flib.lib`, contains functions that a compiled program can call to perform floating-point operations. If you do not use floating-point, you do not have to include this library.

You can also create your own object libraries and link them. The linker includes and links only those library members that resolve undefined references.

9.14.3 Autoinitialization (ROM and RAM Models)

The C compiler produces tables of data for autoinitializing global variables. These tables are contained in a named section called **.cinit**. The initialization tables can be used in either of two ways.

- **ROM Model** (-c linker option)

  Variables are initialized at *run time*. The .cinit section is loaded into memory along with all the other sections. The linker defines a special symbol called `cinit` that points to the beginning of the tables in memory. When the program begins running, the C boot routine copies data from the tables into the specified variables in .bss. This allows initialization data to be stored in ROM and then copied to RAM each time the program is started.

  Figure 9-12 illustrates the ROM autoinitialization model.

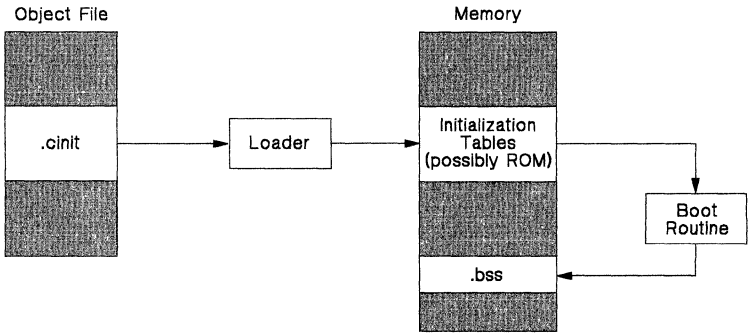


Figure 9-12. ROM Model of Autoinitialization

- **RAM Model** (-cr linker option)

  Variables are initialized at *load time*. This enhances performance by reducing boot time and saving memory that would be used by the initialization tables. (Note that you must use a smart loader to take advantage of the RAM model of autoinitialization.)

  When you use -cr, the linker marks the .cinit section with a special attribute. This attribute tells the linker *not* to load the .cinit section into memory. The linker also sets the symbol `cinit` to -1; this tells the C boot routine that the initialization tables *are not* present in memory. Thus, no runtime initialization is performed at boot time.

  When the program is loaded, the loader must be able to:

  - Detect the presence of the .cinit section in the object file.
  - Detect the presence of the attribute (STYP_COPY) that tells it not to copy the .cinit section.
  - Understand the format of the initialization tables (this is described in the *TMS34010 C Compiler Reference Guide*).

  The loader then uses the initialization tables directly from the object file to initialize variables in .bss.

  Figure 9-13 illustrates the RAM autoinitialization model.

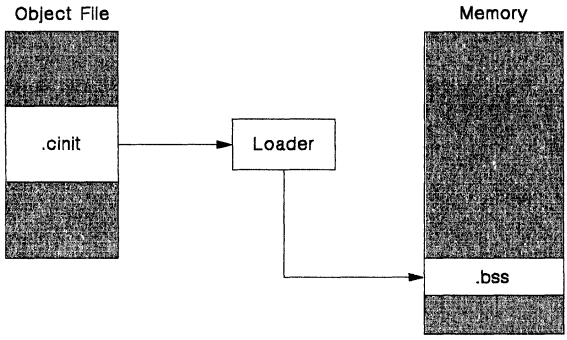


Figure 9-13. RAM Model of Autoinitialization

9.14.4 The -c and -cr Linker Options

The following list outlines what happens when you invoke the linker with the -c or -cr option.

- The symbol `_c_int00` is defined as the program entry point. `_c_int00` is the start of the C boot routine in `boot.obj`; referencing `_c_int00` ensures that `boot.obj` is automatically linked in from the runtime support library `rts.lib`.
- The .cinit output section is padded with a termination record (two bytes filled with 0s) so that the boot routine (ROM model) or loader (RAM model) knows when to stop reading the initialization tables.
- In the ROM model (-c option), the linker defines the symbol `cinit` as the starting address of the .cinit section. The C boot routine uses this symbol as the starting point for autoinitialization.
- In the RAM model (-cr):
  - The linker sets the symbol `cinit` to -1. This indicates that the initialization tables are not in memory, so no initialization is performed at run time.
  - The STYP_COPY flag (010h) is set in the .cinit section header. STYP_COPY is the special attribute that tells the loader to perform autoinitialization directly and not to load the .cinit section into memory. The linker does not allocate space in memory for .cinit.

9.15 Linker Example

This example links three object files named `demo.obj`, `fft.obj`, and `tables.obj` and creates a program called `demo.out`. The symbol `SETUP` is the program entry point.

Assume that target memory has the following configuration:

Address Range:	Memory Contents:
00000 to 1FFFF	Video RAM
20000 to 2FFFF	General-purpose RAM
30000 to 3FFFF	No physical memory
40000 to FFFFFFFF	ROM

The output sections are constructed from the following input sections:

- The .text output section contains executable code. It is constructed from the .text sections of input files `demo.obj` and `labels.obj`, and must be allocated into ROM.
- The .data output section contains initialization data. It is constructed from the .data sections of input files `tables.obj` and `labels.obj`, and must also be allocated into ROM. In addition, the application requires that the .data sections from the two files be separated by 128 bits, all of which are set to 1.
- An output section named `screen` is constructed from the .bss section of `demo.obj`. This section defines the screen memory; it must be linked into video RAM and initialized with the value 8Fh.
- The .bss output section contains the .bss section from input file `labels.obj`. This section defines global and static data; it must be linked into general-purpose RAM, and does not need to be initialized.
- An output section named `int_vecs` is constructed from the `int_vecs` section of `tables.obj`. This section contains interrupt vectors; it must be allocated in ROM at address 0FFFFFFE0h.

Figure 9-14 illustrates the linker command file for this example; Figure 9-15 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
tables.obj
labels.obj

/*************************************************************************/
/****                Specify the Memory Configuration                 ****/
/*************************************************************************/

MEMORY
{
    VIDEO:    origin = 0h,        length = 20000h
    RAM:      origin = 20000h,    length = 10000h
    ROM (R):  origin = 40000h,    length = FFFC0000h    /*  Read only  */
}

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

    .data:                      /* Link the .data sections             */
    {
        tables.obj(.data)
        . += 128;               /* Create a 128-bit hole               */
        labels.obj(.data)
    } = 0FFFFFFFFh >ROM         /* Fill the hole                       */

    screen:                     /* Create a new section for screen mem */
    {
        demo.obj(.bss)
    } = 8F8F8F8Fh >VIDEO

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

    int_vecs 0FFFFFFE0h: { }    /* Link and bind interrupt vectors     */
}

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

Figure 9-14. Linker Command File, demo.cmd

Invoke the linker with the following command:

```
gsplnk demo.cmd
```

This creates the map file shown in Figure 9-15 and an output file called `demo.out` that can be run on the TMS34010.

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

OUTPUT FILE NAME:   <demo.out>
ENTRY POINT SYMBOL: "SETUP"  address:  00040000


MEMORY CONFIGURATION
          name      origin          length          attributes
        --------   --------        ---------        ----------
        VIDEO      00000000        000020000            RWIX
        RAM        00020000        000010000            RWIX
        ROM        00040000        0FFFC0000            R


SECTION ALLOCATION MAP

 output                                  attributes/
section      origin        length         input sections
-------    ----------    ----------      ----------------
screen     00000000      00002000
             00000000      00002000        demo.obj (.bss) [fill = 8f8f8f8f]

.bss       00020000      000000200       UNINITIALIZED
             00020000      000000200       labels.obj (.bss)

.text      00040000      000007110
             000400000     000007110       demo.obj (.text)

.data      00047110      00000b600
             00047110      000003190       tables.obj (.data)
             0004a2a0      000000080       --HOLE-- [fill = ffffffff]
             0004a320      000003190       labels.obj (.data)
             0004a4b0      000003190       demo.obj (.data)

int_vec    fffffe0       000000020
              fffffe0       000000020       tables.obj (int_vecs)


GLOBAL SYMBOLS
 address    name                                 address    name
---------   ----                                ---------   ----
00040000    SETUP                               00000000    extvar
0004006e    cont                                00020200    end
00052710    edata                               00040000    SETUP
00020200    end                                 0004000a    start
00047110    etext                               0004006e    cont
00000000    extvar                              00040104    sub
0004d4b0    list                                00047110    etext
0004d4ce    main                                0004d4b0    list
0004d4ba    p1asm                               0004d4ba    p1asm
0004d4c4    p2asm                               0004d4c4    p2asm
0004000a    start                               0004d4ce    main
00040104    sub                                 00052710    edata

[12 symbols]
```

Figure 9-15. Output Map File, demo.map

Section 10

Object Format Converter Description

Most EPROM programmers do not accept COFF object files as input. The object format converter converts a COFF object file into one of three object formats that most EPROM programmers accept as input:

- **Tektronix hex object format:** This format supports 32-bit addresses.
- **Intel hex object format:** This format supports 16-bit addresses.
- **TI-tagged object format:** This format supports 16-bit addresses.

> **Note:**
>
> If your code uses addresses that are significant to more than 16 bits, use the Tektronix format.

The object format converter accepts one COFF object file as input. If you are converting to TI-tagged object format, the utility produces one output file. If you are converting to Tektronix or Intel object format, the utility produces two output files: one output file contains the high (most significant) bytes, and the other file contains the low (least significant) bytes.

This section contains the following topics:

10.1 Object Format Converter Development Flow

Figure 10-1 illustrates the object format converter's role in the assembly language development process.

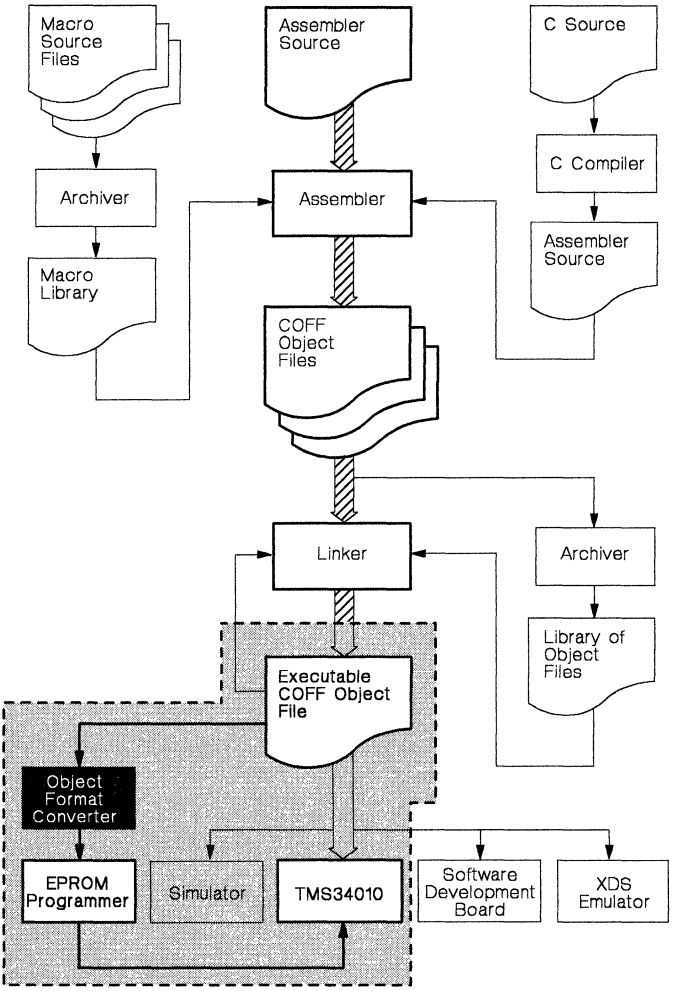


Figure 10-1. Object Format Converter Development Flow

10.2 Invoking the Object Format Converter

To invoke the object format converter, enter:

gsprom *[-option] [COFF input file [output file$_1$ [output file$_2$]]]*

gsprom is the command that invokes the object format converter; all parameters are optional. The *options* can be entered anywhere on the line, but the order of filenames is significant. The filenames (if used) are interpreted as:

1) The input filename,

2) The output filename (for TI-tagged format) or the high-byte output filename (for Tektronix or Intel format), **and**

3) The low-byte output file (for Tektronix or Intel format). (If you specify TI-tagged format and name a third file, the utility ignores the third filename.)

- There are three options:

  -i specifies Intel hex object format for the output.

  -t specifies TI-tagged object format for the output.

  -x specifies Tektronix hex object format for the output.

  If you don't specify an option, the object format converter produces Tektronix hex format output files.

- If you do not specify an input filename, the object format converter prompts for it. If you specify a filename without an extension, the utility assumes that the filename has a default extension of ***.obj***.

- If you do not specify a second filename, the object format converter uses the input filename with an extension of:

  - ***.tag*** (for TI-tagged format) **or**
  - ***.hi*** (for Tektronix or Intel format).

- If you do not specify a filename for the low-byte output file but *do* specify a name for the high-byte output file, the object format converter uses the high-byte file name with an extension of ***.lo***. If you do not specify a filename for the low-byte file *or* the high-byte file, the object format converter uses the input filename with an extension of ***.lo***.

When the utility finishes converting the input file, it prints the message `Translation complete`.

10.3 Object Format Converter Examples

Here are some examples of using the object format converter.

- **Example 1:**

  You can invoke the object format converter with no options and no filenames:

  ```
  gsprom
  ```

  The utility will print the following banner and prompt:

  ```
  COFF Object Converter     Version 5.01, 87.610
  (c) Copyright 1987, Texas Instruments Inc.

      Coff file [.obj]:
  ```

  If, for example, you respond to the prompt with a filename of `fft`, the object format converter uses the file `fft.obj` as an input file. The utility produces two output files named `fft.hi` and `fft.lo` in Tektronix hex format. (Tektronix format is the default when you don't specify a format.)

- **Example 2:**

  If you enter:

  ```
  gsprom -i in out1 out2
  ```

  the utility uses `in.obj` as the input file. It creates two Intel hex format files named `out1.hi` and `out2.lo`.

- **Example 3:**

  If you enter:

  ```
  gsprom -x in.tmp out.x
  ```

  the object format converter uses `in.tmp` as the input file. It produces Tektronix hex format output; the high-byte file is named `out.x`, and the low-byte file is named `out.lo`.

- **Example 4:**

  If you enter:

  ```
  gsprom -t test
  ```

  the object format converter uses `test.obj` as the input file. It produce an output file named `test.tag` in TI-tagged format.

There are two situations in which the object format converter aborts execution:

1) If any of the specified files cannot be opened, the object format converter prints the message `Input COFF file cannot be opened` and aborts.

2) If you supply the utility with the name of an invalid object file, the object format converter prints the message `Corrupt input file` and aborts.

Section 11

Simulator Description

The TMS34010 simulator is a debugging tool that provides software simulation of the TMS34010 hardware functions and of a configurable graphics environment. The simulator command set displays and maintains graphics and machine status information, and controls execution of the software system under development.

The simulator can be used to design, implement, and evaluate both graphics and nongraphics software systems. The simulator:

- Allows complete control over the simulation status
- Recognizes the entire TMS34010 assembly language instruction set
- Uses memory efficiently to simulate both TMS34010 program memory and screen memory
- Simulates the host interface, memory controller, and cache operation
- Supports breakpoints and traces on a variety of memory accesses
- Displays the machine state in a screen-oriented format
- Provides a versatile methods of command entry, with error reporting, file input, and multiple command buffers
- Provides statistical information that allows you to evaluate program performance, including the cache hit/miss ratio

The simulator's relative execution speed depends on the host system; MS-DOS and PC-DOS systems generally operate 20,000 times slower than the TMS34010.

Topics in this section include:

11.1 Simulator Development Flow

Figure 11-1 illustrates the simulator's role in the assembly language development flow. The simulator accepts linked COFF object files as input.

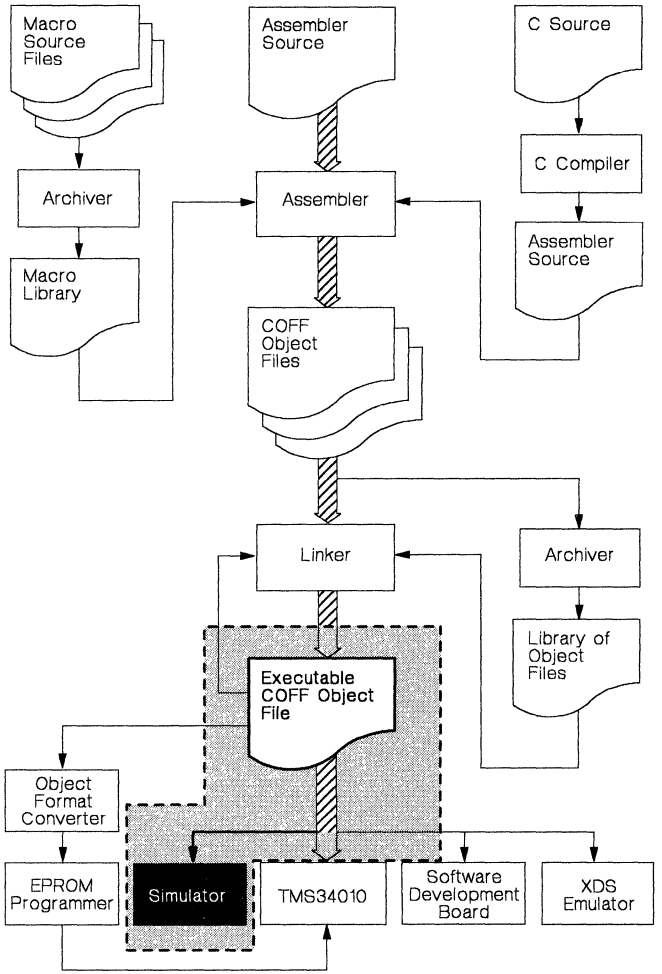


Figure 11-1. Simulator Development Flow

11.2 Invoking the Simulator

To invoke the simulator, enter one of these two commands:

IBM-PC: **gspsim** *[-options] [object file [offset]]*

TI-PC: **gspsimt** *[-options] [object file [offset]]*

gspsim is the command that invokes the simulator on an IBM-PC.

gspsimt is the command that invokes the simulator on a TI-PC.

options are optional parameters. When used, options must be preceded by a hyphen; bad options are ignored. Valid options include:

- **-f** tells the simulator to expect input from the command input file `gspinput.000`.
- **-t** disables the special trap functions that the simulator normally associates with traps 27, 28, and 29.

object file is an optional parameter. It specifies a COFF object module that the simulator loads and executes following invocation.

offset is an optional parameter; you can use it when you specify a load file. The simulator relocates the object module by adding the offset to all the relocation entries in the module.

The simulation session begins after you invoke the simulator; the simulator:

1) Displays its banner,
2) Turns on the graphics card,
3) Loads the object module (if specified),
4) Performs a reset, **and**
5) Displays the initial screen shown in Figure 11-2.

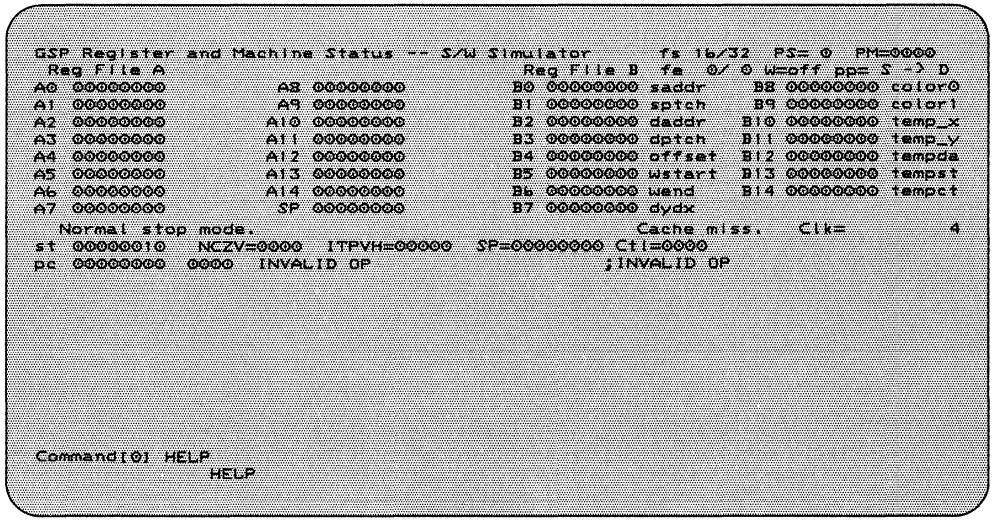


Figure 11-2. Initial Simulator Display

11.3 Hardware and System Requirements

The simulator runs on 8088, 8086, and compatible derivatives (using MS-DOS versions 2.11 and higher or PC-DOS versions 2.1 and higher). MS-DOS systems that are directly supported include:

- The IBM-PC, PC/XT, PC/AT, and compatible machines with 512K bytes of memory and CGA emulation.
- The TI-PC with 3-plane color graphics support and 512K bytes of memory.

System requirements for operating the simulator include:

- A host operating and display system as described above.
- An editor for manipulating TMS34010 assembly language and source files.
- The TMS34010 assembler and linker (and optionally the TMS34010 C compiler) for creating object files.

In addition, you need a working knowledge of the TMS34010 instruction set.

On an IBM-PC, you must have the following `CONFIG.SYS` file:

```
BUFFERS = 20
FILES = 20
DEVICE = C:/MSDOS/ANSI.SYS
```

A copy of this file is shipped with the simulator. Your system must be configured so that the `CONFIG.SYS` file can install the `ANSI.SYS` device driver when the system is booted.

We suggest that you create a directory called `\GSPTOOLS` to contain the simulator and the simulator help files. Use the DOS SET command to equate the `\GSPTOOLS` pathname with the name `GSPDIR`; for example,

IBM-PC: `SET GSPDIR=C:\GSPTOOLS`

TI-PC: `SET GSPDIR=E:\GSPTOOLS`

(Be sure to enter the command exactly as shown – with uppercase letters and without blanks except as shown.) This allows the simulator to access the help files from other directories.

11.4 Screen Displays

The TMS34010 simulator displays two separate sets of information:

- The **machine-state display**, which provides a complete set of status and statistical information.
- The simulated **graphics display** (also called the graphics environment).

11.4.1 Machine-State Display

The simulator updates the machine-state display when it executes commands. Figure 11-3 shows the initial machine-state display. Each field in this illustration is numbered; the numbers are keyed to the list that follows the illustration.

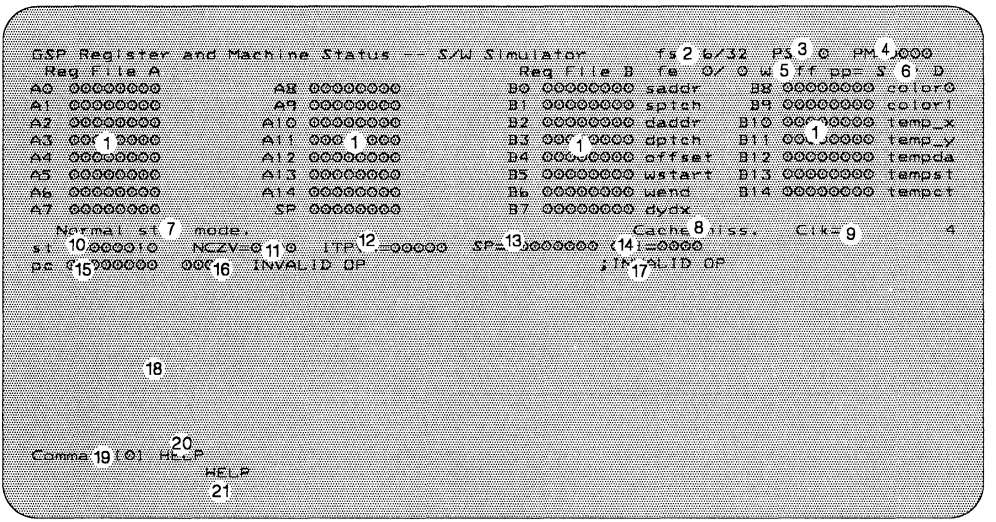


Figure 11-3. Simulator Display Format

1) **Register Display.** This area displays the TMS34010 internal registers. You can display two sets of registers:

- The general-purpose register files (A and B) and the stack pointer **or**
- I/O registers.

The general-purpose registers are the default display. During graphics operations, the B-file registers have special functions; these functions are listed with the registers in the display. When the simulator lists the I/O registers, it displays their functions and their addresses.

The DR command toggles the display between the general-purpose registers and the I/O registers. For an example of the I/O register display, see the DR command.

2) **Field Information:** This area of the screen (**FS** and **FE**) displays the sizes of fields 0 and 1, respectively, followed by their respective field extension bits (FE0 and FE1). The first values for FS and FE apply to field 0; the second values apply to field 1. All four of these values are extracted from the status register; they are displayed as decimal numbers.

   FS0 is selected independently of FS1. Status bits 0–4 select field size 0, bits 6–10 select field size 1. Valid field sizes range from 0 to 32 bits.

   FE0 is selected independently of FE1. Status bit 5 selects the field extension type of field 0, bit 11 selects the field extension type of field 1. A value of 0 indicates zero extension, 1 indicates sign extension.

   The field size and field extension used by the current instruction are highlighted in green. If the current instruction does not use a field size or extension, then both sets of values are displayed in yellow.

3) **Pixel Size:** This area of the screen (**PS**) displays the current pixel size (this is the value in the PSIZE register). Valid pixel sizes include 1, 2, 4, 8, and 16 bits per pixel.

4) **Plane Mask:** This area (**PM**) displays the value of the plane mask (this is the current value of the PMASK register).

5) **Windowing Option:** This area (**W**) displays the current windowing option, which is selected by setting bits 6 and 7 of the CONTROL register. Valid windowing options include:

   W = off windowing is not enabled.
   W = int any attempt to write inside the window generates an interrupt, and no pixels are drawn.
   W = on if the object lies entirely within the window, it's drawn, otherwise no pixels are drawn.
   W = pick (also called window *clip*.) The portion of the object that lies inside the window is drawn.

   If a window option other than *off* is selected, it is highlighted in cyan.

6) **Pixel Processing Option:** This area of the screen (**PP**) lists the currently selected pixel processing option, which is selected by bits 10–14 of the CONTROL register. The S→D pixel processing option is the default; any other option that is selected is highlighted in cyan. For a list of valid pixel processing options, see the PP simulator command.

7) **Monitor Status Messages:** This area displays the current status of the TMS34010 simulator; the messages are self explanatory.

8) **Cache Status:** This area of the screen indicates whether the last instruction executed was in cache. Messages that may be displayed include:

Cache off	the cache is not enabled.
Cache miss	the cache is enabled but does not contain the currently executing instruction.
Cache hit	the cache is enabled and contains the currently executing instruction.

Cache operation is simulated during execution and reverse-assembly. If, when cache is enabled, you display a memory location to which the PC is pointing, you may not see a change in the machine-state display and in the actual opcode that is executed. This is faithful to actual TMS34010 operation with the cache enabled. To avoid this situation, use the CF command to flush the cache or the CD command to toggle the CD bit in the CONTROL register.

The simulator maintains a record of cache statistics when the cache is enabled. These statistics include:

- A total count of cache accesses **and**
- The ratio of hits to misses (listed as a percentage).

Use the DC (display cache) command to display the cache status in the scratch area. The CLCS (clear cache statistics) command resets these statistics to 0.

9) **Clock Timing:** This area (**CLK**) displays the number of clock cycles consumed thus far in the instruction simulation. The clock value is based on instruction execution times and memory accesses. This clock value does not represent the amount of time consumed by the TMS34010 system for video timing, RAM refresh, or shift register access. The initial clock cycles value is 4, for the 4 clock cycles consumed by fetching the PC from the trap 0 vector for reset. One clock cycle is 160 ns.

10) **Status Register:** This area (**ST**) displays the status register contents.

11) **Status Elements:** This area displays the current values of four status bits contained in the status register. These include:

N	sign bit
C	carry bit
Z	zero bit
V	overflow bit

12) **Control Elements:** This area displays the current values of five bits contained in the status and CONTROL registers, including:

I refers to the IE (interrupt enable) bit in the status register. Setting IE=0 disables all maskable interrupts; setting IE=1 enables all maskable interrupts.

T refers to the transparency bit in the CONTROL register. Setting T=0 disables transparency; setting T=1 enables transparency.

P refers to the PBX (PIXBLT executing) bit in the status register. The TMS34010 sets this bit when a PIXBLT or FILL instruction is interrupted. PBX=1 indicates that the interrupt occurred on an instruction boundary; PBX=1 indicates that the interrupt occurred during instruction execution.

V refers to the PBV (PIXBLT vertical direction) bit in the CONTROL register. This bit determines the vertical direction for PIXBLT instructions that use XY addressing. Setting PBV=0 increments the Y dimension (move from top to bottom); setting PBV=1 decrements the Y dimension (move from bottom to top).

H refers to the PBH (PIXBLT horizontal direction) bit in the CONTROL register. This bit determines the horizontal direction for PIXBLT instructions that use XY addressing. Setting PBH=0 increments the X dimension (move from left to right); setting PBH=1 decrements the X dimension (move from right to left).

13) **Stack Pointer:** This area (**SP**) displays the contents of the stack pointer.

14) **Control Register:** This area (**CTL**) displays the contents of the CONTROL register.

15) **Program Counter:** This area (**PC**) displays the current contents of the program counter.

16) **Data at the Location Pointed to by the PC.** The area following the PC displays the value of the word that the PC is pointing to and the word's reverse-assembly.

17) **Last Instruction Executed.** This area contains the reverse-assembly of the last instruction that was executed. If this instruction is the same as the current instruction, it is displayed in green; otherwise, it is displayed in cyan. Specific error messages and breakpoint halt conditions are also displayed in this area.

18) **Graphics and Scratch Display Area** (10 lines). This is the scratch display area. Information is listed in the 10 blank lines between the reverse-assembly line and the command line.

19) **Command Prompt and Current Buffer.** This line prompts for command entry and displays the current buffer. The simulator maintains ten command buffers (0–9) for storing commands before or after their execution.

20) **Command Entry Line.** This is where you enter commands. Except for menu-driven commands, information is only entered from the command line in the space following the command prompt. Commands are displayed in uppercase; even if you enter them in lowercase, the simulator immediately translates them to uppercase. The simulator cursor is a shaded, blinking rectangle; in this document, it is represented by the character _.

21) **Last Command Entered.** The simulator echoes the last command entered by displaying it beneath the command line.

11.4.2 Displaying Graphics and Status Information Simultaneously

The graphics display and the machine-state display can be viewed separately or simultaneously on monitors with independent text and graphics displays (such as the TI-PC). In this situation, the machine-state and graphics displays can be independently toggled, producing four possible combinations of screen information:

- Graphics display with machine-state display superimposed (default)
- Machine-state display only
- Graphics display only
- Blank screen with command line

On monitors with interdependent text and graphics, these two sets must be viewed alternately:

- Machine-state display only (default)
- Graphics display only

You can use the TX and GR commands to swap display modes.

11.4.3 Using the HELP Function

You can type HELP or H at any time to display the help utility menu. The help utility contains a menu that lists classes of instructions. Invoking any of these calls up a file on the host that contains reference information about the commands.

> **Note:**
>
> The help files must be:
>
> 1) In the current directory on the current disk **or**
>
> 2) In the directory specified with `GSPDIR` (see Section 11.3 on page 11-4).

The help files contain an initial listing followed by a brief summary of each commands syntax and action. Figure 11-4 (page 11-10) illustrates the help menu.

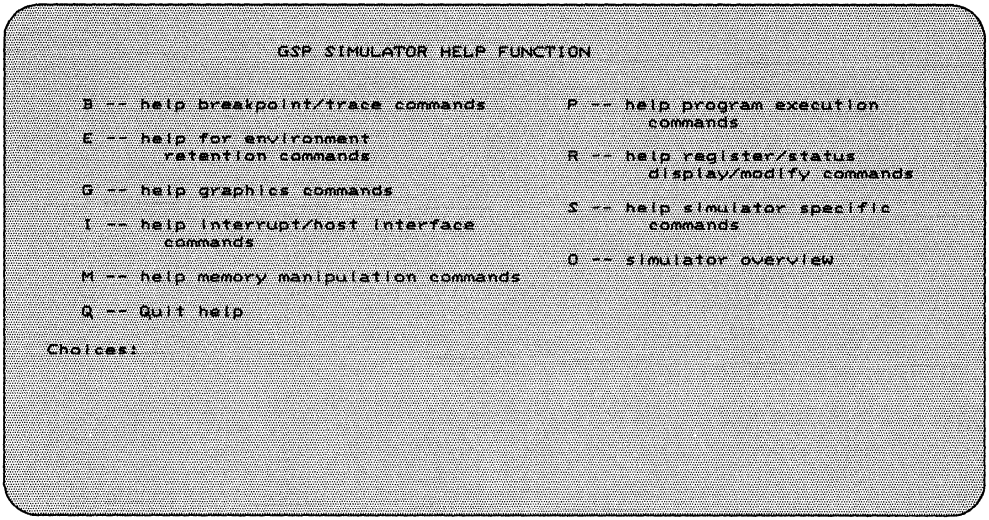


Figure 11-4. Simulator Help Menu

11.5 Entering Commands

The simulator supports a set of general commands as well as TMS34010-specific commands.

Initially, all commands are entered on the command line, as shown in Figure 11-3 on page 11-5. You can enter commands in uppercase or lowercase, or in any combination; the simulator converts them to uppercase before it interprets them. As you enter commands, the simulator places them in the currently active command buffer to be executed. The command line is the line that contains the prompt:

```
Command[0]
```

In this example, the number **0** indicates that command buffer 0 is the active buffer. The prompt and cursor generally appear as:

```
Command[1] _
```

(_ shows the cursor position.) The command line contains the most recent command entered into the command buffer. You can type over it or edit it to enter a new command. The line beneath the command line echoes the most recently stored command; you cannot edit this line, but the simulator updates it as commands are executed or stored.

The simulator allows you to edit the command line with some simple editing keys:

- **Backup one character.** Use one of the following:
  - ← key,
  - *backspace* key,
  - *control-S* keys, **or**
  - *control-H* keys.
- **Forward one character.** Use one of the following:
  - → key **or**
  - *control-D* keys.
- **Forward one word.** Use one of the following:
  - *tab* key **or**
  - *control-F* keys.
- **Backward one word.** Use one of the following:
  - *shift-tab* keys **or**
  - *control-A* keys.
- **Delete character.** Use one of the following:
  - *delete* key **or**
  - *control-G* keys.
- **Insert characters.** Use one of the following:
  - *insert* key **or**
  - *control-V* keys.

After you type a command name (and edit it, if necessary), type either a *linefeed, control-J,* or ***CR***. (carriage return) so that the simulator can process the command:

- ***CR*** truncates the command, and executes the portion of the command to the left of the cursor.
- *linefeed* and *control-J* execute the entire command line.

You can re-execute a command by placing the cursor in the leftmost position and entering a carriage return. You can enter multiple commands on one line by separating them with semicolons.

11.5.1 Command Parameters

Some simulator commands have numeric parameters; you can use decimal or hexadecimal numbers. Numeric parameters default to the format that is most often used for a particular parameter. For example, addresses default to hex, register numbers default to decimal, and register contents default to hex.

You can override default formats by using one of these characters:

% (prefix) specifies decimal format.

H (suffix) specifies hexadecimal format.

You can also specify a negative number (whether or not you are overriding the default format) by prefixing it with a minus sign (-).

Here are some examples of that use numeric parameters. These examples use the A command to modify or display the contents of an A-file register. The syntax for the A command is:

A*n register value*

(*n* is a register number in the range 0–14.) The default format for the value is hexadecimal. All of the following commands set the contents of register A12 to 0FFFFFFFFh.

```
Command[1] A12 FFFFFFFF
Command[1] A%12 FFFFFFFF
Command[1] ACH FFFFFFFF
Command[1] A12 -1
Command[1] A12 -%1
Command[1] ACH -1H
```

> **Note:**
>
> All memory references are specified as **bit** addresses. Thus, the last ASCII character of all word-aligned addresses is 0. The last four bits of memory addresses that are specified on the command line are forced to 0, unless otherwise stated.

You can also use a register as a numeric parameter to specify a 32-bit value. The format for this type of parameter is **R***n*; *n* can be:

- **A0–A14** (an A-file register)
- **B0–B14** (an B-file register)
- **SP** (the stack pointer),
- **PC** (the program counter), **or**
- **ST** (the status register).

You can also use the X half (16 LSBs) or Y half (16 MSBs) of a register to specify a 16-bit numeric parameter. The formats for these types of parameters are **RX***n* and **RY***n*.

For example, you can load register A12 with the the contents of the status register:

```
Command[1] A12    RST
```

You can load the stack pointer with the contents of the program counter:

```
Command[1] SP     RPC
```

You can use the MM command to change the contents of a single word that the SP points to:

```
Command[1] MM     RSP   0FFFFh
```

You can also use the *R* prefix to access A- and B-file registers. The first example below negates the contents of register A14. The second example modifies the two words pointed to by the SP with the contents of register B2. The third example modifies the contents of a single word pointed to by register B8 with the Y half of register A1.

```
Command[1] A14    -RA14
Command[1] MM     RSP   RB2
Command[1] MM     RB8   RYA1
```

11.5.2 Command Buffers

The simulator maintains 10 command buffers for storing commands before or after their execution. The active command buffer is indicated by the value inside of the square brackets following the `Command` prompt. The initial default buffer is 0, indicated by:

```
Command[0].
```

You can select a command buffer by entering one the following as the first character on the line:

- A number (0–9) selects a specific buffer.
- + or ↑ advances to the next buffer.
- - or ↓ goes to the previous buffer.

Changing to a new buffer does not change the contents of the current buffer.

You can use multiple command buffers to store specific commands. This allows you to execute a chain of commands with fewer keystrokes. Command buffers can be chained together to provide lengthy command sequences. To

chain several buffers together, specify the next buffer to be executed as the last command on the command line:

```
Command[0] SS; DM 0 200; 1
```

This example single-steps, displays memory from address 0 to address 200, and then executes the contents of buffer 1. A buffer can even reference itself to provide a simple looping mechanism.

```
Command[0] A 13 340990BC
           A 13 340990BC

Command[0] 5 CR13 340990BC
         A       13 340990BC

Command[5] HELP
           HELP
```

In this case, buffer 0 still contains the command `A 13 340990BC` which can be re-executed by returning to buffer 0.

You can use the +, -, ↑, and ↓ keys to move through the buffers:

```
Command[0] A 13 340990BC
           A 13 340990BC

Command[0] + CR13 340990BC
           A       13 340990BC

Command[1] HELP
           HELP

Command[1] - CRELP
           HELP

Command[0] A 13 340990BC
           A 13 340990BC
```

Buffer 0 still contains the command `A 13 340990BC`, which can be re-executed by returning to buffer 0.

You can store a string of commands in a buffer without executing them by preceding the string with an ! character:

```
Command[1] !A 13 45; A14 6000; run CR
             A9 801AC

Command[1] A 13 45; A14 6000; run
             A 13 45; A14 6000; run
```

11.5.3 Loading and Running Code

Use the L command to load object code into the simulator. The syntax is:

L *filename [offset]*

This command provides the object file name and an optional signed program memory offset.

The following example loads file `coda.out`.

```
Command[1] L CODA.OUT 10200
```

The simulator will open `coda.out`, read its contents, interpret it, and load it into the simulated memory using the offset 00010200h. Note that this value offsets any addresses given in the object module.

You can offset any object code file by specifying the offset value as a parameter of the L command. Figure 11-5 describes protected (allocated) areas of memory. Beware of offsetting nonrelocatable files.

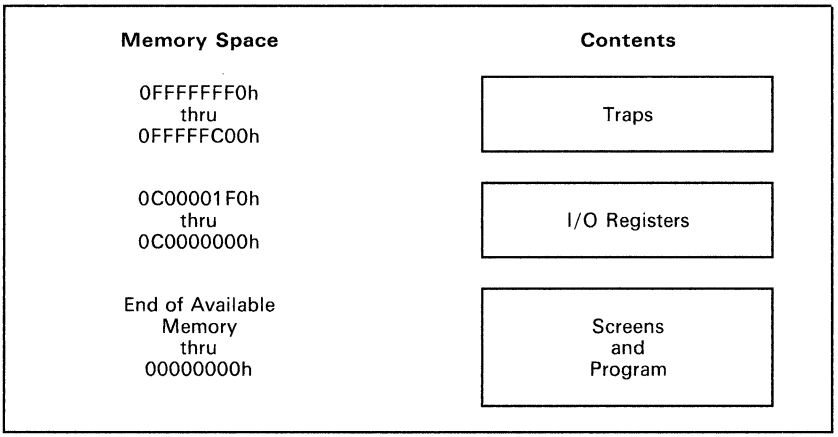


Figure 11-5. Dedicated and Available TMS34010 Memory Spaces

Note that the simulator cannot check to see if screen or program boundaries are violated. Screen and program spaces are distinguished only by how you treat them; therefore, any area of available memory can be used for either purpose. If you attempt to write to a memory location outside of the defined memory range, the simulator issues an error message and halts.

11.5.4 Line Assembler

The simulator can assemble single lines of TMS34010 assembly language code. It places the resulting opcodes and operands into memory via the MM command. The line assembler accepts only absolute numeric values.

11.5.5 Error Reporting

Errors are reported in several ways:

- If you enter a command or parameter incorrectly, the simulator displays an error message on the line beneath the command line. These error messages are usually diagnostic (for example, they can show you why an entry was incorrect). For example,

  ```
  Command[1] && 4@ CR
        A 9 34010

  Command[1] && 4@
  Command not recognized; re-enter

  Command[1] && 4@
        && 4@
  ```

  Command error messages of this sort are queued (stored up), and remain available for viewing until the list is exhausted. You can clear a message by typing *ESC*; each message requires a separate *ESC* before control returns to the command line. The simulator does not remove the incorrect command from the buffer; this allows you to edit and re-enter it.

- Instruction execution errors are displayed in red in the scratch display, and remain displayed until you clear them (use CLS) or until other information is displayed.

11.6 System Simulation

The simulator's chief purpose is to accurately simulate the TMS34010 instruction set. Where there are deviations, they have been made in the interest of providing a more useful simulation tool (for example, traps 29–27 have reserved functions). The simulator does not simulate the video timing, RAM refresh, and shift register operations implied by I/O registers.

11.6.1 Local Memory Simulation

The memory that is available for simulated local memory is determined by the memory available on the host. The definition (beginning and ending addresses, on-screen and off-screen memory) can be specified with the M command. Memory is accessed by the program counter, the host interface, and the TMS34010 instruction set via the cache and memory controller simulation. These accesses to memory can be monitored via the trace (TR) and breakpointing (BP) features of the simulator. Up to 20 trace and/or breakpoint definitions are allowed at any one time. You can also access memory directly via simulator commands (F – fill memory, MM – memory modify, DM – display memory, and IO – display/modify I/O registers, etc.) in which case trace and breakpointing are inhibited.

The graphics display (when enabled) reflects changes that affect on-screen memory, except when a program is downloaded.

11.6.2 Interrupts Simulation

The simulator simulates the TMS34010 interrupt structure through a a timing schedule. Use the I command to define this schedule; you can supply a count and a frequency of occurrence selection for each interrupt (including RESET, INT1, INT2, INT3, NMI, HOST, DISP, and WV).

Interrupts are handled when they occur in a priority fashion and index into the trap and interrupt vector table for the beginning of the interrupt handling routine. The INTPEND and INTENB registers and their effect on the handling of interrupts are also simulated.

11.6.3 Host Interface Simulation

The simulator simulates the host interface by means of:

1) A host input file (HIF) that you provide, **and**
2) A timing schedule that you provide in the form of a count and a frequency of occurrence selection (given as the number of clock cycles between inputs).

The IH (initiate host) command allows you to initiate and control host inputs by selecting, before executing code, the number of inputs and their frequency of occurrence. For instance, you could specify that you want a host input to occur every 96 clock cycles; the simulator would then generate an input every 96 clock cycles for the specified number of times. The data in the host input file controls the action that the simulator takes at input time. This continues until either the count or the data is exhausted.

11.6.3.1 Host Input File Layout

If you want to simulate host inputs, you must first create a host input file (HIF). This file is named `host.in`, according to the conventions described by the IH command. Each line of the HIF contains the information needed to simulate a single exchange between the TMS34010 and the host. You use the IH command to provide the simulator with initial access to the HIF. Access to the HIF is driven by the TMS34010 clock and occurs automatically as the clock is advanced.

11.6.3.2 Host Input File Format

Each line of the HIF contains the HFS0, HFS1, HREAD, and HWRITE pin values ($\overline{\text{HCS}}$, chip select, is assumed active). If the operation to be performed by the host is a write, the field following the pin values beginning in column 6 contains the 16-bit data value which is to be placed on the HD bus. If the host wants to read, then the resulting value on the HD bus is written out to the host output file (HOF), `host.out` with its address and the clock count at which the read occurred. The number of times and frequency of occurrence of host interrupts is controlled by the interrupt control commands for the nonmaskable interrupt. Note that both HIF and HOF are ASCII text files of 80 columns per record.

Table 11-1. HIF ASCII Record Format

Columns	Info	Description
1	HFS0	Desired value for HFS0 pin (0 or 1).
2	HFS1	Desired value for HFS1 pin (0,1).
3	HREAD	Desired value of HREAD pin (0,1).
4	HWRITE	Desired value of HWRITE pin (0,1).
5	xxxxx	Ignored
6–9	HData	The sixteen bit data word on the HD bus in hexadecimal. This data is used only if the host is writing to the TMS34010.
10–80	don't care	Information in this block is disregarded – it's a good comment area.

See the IH command for information on how to set up a host input scenario.

11.6.4 Graphics Simulation

The simulator displays graphics on the host screen (depending on mode selection and host configuration) as an offshoot of the TMS34010 local memory simulation.

The default graphics display control parameters (modified using the G command) are such that TMS34010 memory from TMS34010 address 00002000h to the end of available screen memory is defined as on-screen and is positioned in the lower left portion of the host screen. This ensures that the major portions of the machine-state display remains free from competing graphics display. On systems with interdependent text and graphics, the definition is the same except that the on-screen memory is placed in the upper left hand corner of the host screen. The default graphics display definition can be recalled after modification by using the GRI command.

You can customize the graphics environment for a particular screen configuration by using the G command. You can customize the graphics environment so that large on-screen memory simulation can be brought to the host screen by simply redefining the location of the host screen corner within memory. Via the simulator monitor, you can control the origin of the screen, as well as the "window" into the screen. The screen, however, is only modified when memory is written to or when you invoke the RS (regenerate screen) command.

The maximum graphic display area is determined by the host system. The TI-PC 3-plane graphics card displays 720-by-276 pixels. You can select a given portion of this screen to view at any one time, and thus simulate a larger screen area than the physical device can display at one time.

On the TI-PC color graphics system, and on all limited display devices, pixel values greater than the maximum number of bits per pixel available on the display device (3 on the TI-PC) are truncated so that only the LSBs are significant. The most significant bits are masked off and the lower bits determine the displayed pixel color according to the device color mapping.

On the TI-PC, then, only pixel sizes of 1 or 2 bits are fully supported, while 4-bit pixel size still generates a graphic display using the three low-order bits of the pixel. Table 11-2 shows the color mapping for the TI-PC system.

Table 11-2. TI-PC Color Mapping

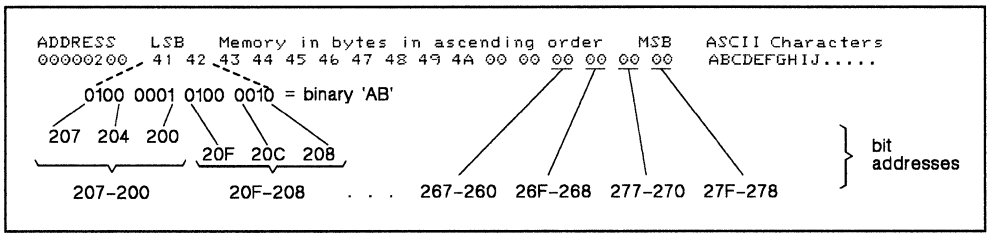
Pixel Value		Color
Decimal	Binary	
0	x000	Black
1	x001	Blue
2	x010	Red
3	x011	Purple
4	x100	Green
5	x101	Cyan
6	x110	Yellow
7	x111	White

11.6.5 DB, DM, and DW Display Comparison

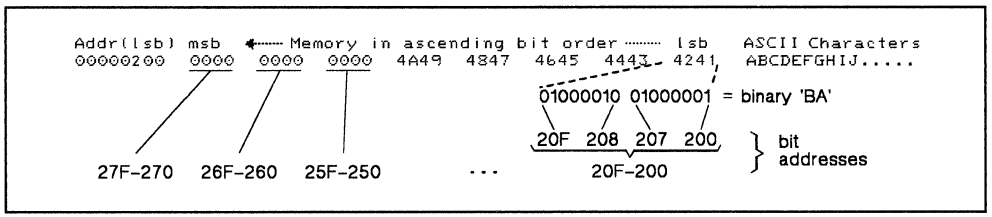
The simulator provides three commands that allow you to display memory in various formats:

DB displays memory in byte format.
DM displays memory.
DW displays memory in word format.

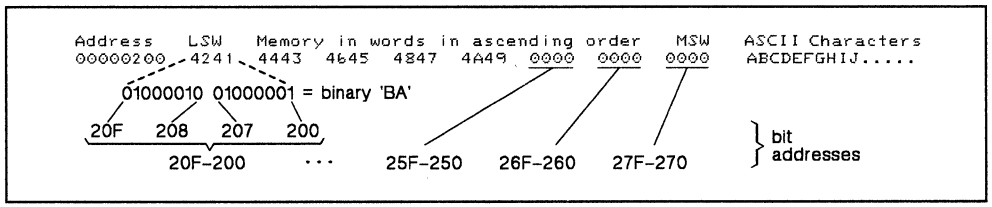
The displays for each of these instructions are different, as Figure 11-6 shows.



(a) DB - Display Byte Command



(b) DM - Display Memory Command



(c) DW - Display Word Command

Figure 11-6. DB, DM, and DW Displays

Similarities among the displays include:

- Each has an 8-digit (32-bit word) address on the left.
- Each displays the memory values in the center.
- Each has recognizable ASCII characters on the right.

The commands differ in the manner in which they display data:

- The DB command displays data in 8-bit values.
  - The least significant byte is on the left.
  - The most significant byte is on the right.
  - The least significant address bit is the rightmost bit (LSB) of the first byte displayed (on the left).

- The DM command displays data in 16-bit words.
  - The least significant byte is on the right.
  - The most significant byte is on the left.
  - The least significant address bit is the rightmost bit on the line.

- The DW command displays data in 16-bit words.
  - The least significant word is on the left.
  - The most significant word is on the right.
  - The least significant address bit is the rightmost bit of the LSW.

11.6.6 Saving Simulator Status

You can use one of several commands to save the current TMS34010 simulation state locally or to a file.

SIO Saves I/O registers.

SR Saves general-purpose registers.

RIO Restores I/O registers locally.

RR Restores registers locally.

RMI [nnn [offset]]
Restores a memory image from `smifil.nnn` applying address offset.

RMS [nnn]
Restores the machine status from `smsfil.nnn`.

SMI start-address end-address [nnn]
Saves a memory image from val1 to val2 in file `smifil.nnn`.

SMS [nnn]
Saves TMS34010 status to file `smsfil.nnn`.

11.7 Demonstration Program

The simulator disks that are shipped as part of the TMS34010 assembly language tools package contain a demonstration program that runs on the simulator. Here are instructions for running the demonstration program.

1) Invoke the simulator.

   IBM-PC: **gspsim** ***CR***

   TI-PC: **gspsimt** ***CR***

2) Restore the graphics environment.

   The demonstration program was written to run in a specific graphics environment. The product disks contain a file that restores this environment. Enter:

   `Command [0] RGE` ***CR***

   This loads the file `sgefil.000`.

3) Clear the memory that the demonstration program uses. Enter:

   `Command [0] F 0 1FFF0 0` ***CR***

   This fills the memory range 0–1FFF0 with 0s.

4) Load the demonstration program.

   IBM-PC: `Command [0] L TUTOR_C.OUT` ***CR***

   TI-PC: `Command [0] L TUTOR_TI.OUT` ***CR***

5) Now run the program. Enter:

   `Command [0] RUN` ***CR***

   The demonstration program draws on a simulated screen; the borders of this screen are drawn for reference. The simulated screen is limited to a size of 256 × 128 pixels; this prevents the graphics demonstration from overwriting the machine-state display.

6) The demonstration program will run until it encounters a software breakpoint (trap 29). When this happens, enter a carriage return to continue execution.

   The demonstration program illustrates several TMS34010 features. You can step through the entire program by entering ***CR*** each time the program encounters a breakpoint. Alternatively, you can execute a specific routine. Table 11-3 lists the program addresses of the routines in the demonstration program.

Table 11-3. Addresses of Routines in the Demonstration Program

Address	Routine
20320h	Pixel transfer (PIXT)
203C0h	Draw and advance (DRAV)
20460h	FILL
20560h	Pixel block transfer (PIXBLT)
207E0h	Pixel processing options and transparency
20850h	Windowing
208F0h	Text (kerning)

To run a specific portion of the demonstration program, load the PC with the desired address and then enter the RUN command. For example, to run the FILL routine, enter the following commands:

```
Command[1]: PC 20460 CR
Command[1]: RUN CR
```

You can use any of the simulator commands to modify or display values while the demonstration program runs. This interrupts the program; enter the RUN command when you're ready to continue.

> **Note:**
>
> To quit execution and leave the simulator, enter Q ***CR***.

11.8 Simulator Commands

Table 11-4 lists according to functional groups. Following Table 11-4, the simulator commands are described in alphabetical order.

Note that command and option abbreviations are indicated by uppercase letters. For example, RUn shows that you can enter this command as RUN or RU.

Table 11-4. Simulator Command Summary

Program Execution Commands	
Command and Syntax	**Operation Description**
CIF	Close input file
CLK *[value]*	Modify or display clock value
Help	Enter help utility
L *filename [offset]*	**Load COFF file**
LE	Display last errors
LH	Display last halts
LM	Display last monitor messages
Q[*][C][S]	Quit simulator
RC *[clock-count]*	Run for specified clock cycles
REset	Reset TMS34010
RUn *[instruction-count]*	Run a program
SS[F][U] *[instruction-count]*	Single step through a program, with or without Fast update and/or Unassembly options
SWitch	Switch command input context
U *[start-address] [end-address]*	Unassemble a program
Z	Zero clock counter
Register Commands	
Command and Syntax	**Operation Description**
A	Display A-file registers
An *[32-bit-value]*	Modify or display an A-file register
B	Display B-file registers
Bn *[32-bit-value]*	Modify or display a B-file register
CLA	Clear A-file registers
CLB	Clear B-file registers
CLIO	Clear I/O registers
CLR	Clear both A- and B-file registers
CTL *[16-bit-value]*	Modify or display CONTROL register
DR	Display A- and B-file registers
IO	Display I/O registers
IOn *[value]*	Modify specified I/O register
NR *register name*	Name a register
PC *[32-bit-value]*	Modify or display program counter
PM *[16-bit-value]*	Modify or display PMASK register
RIO	Restore temporary copy of I/O registers

Table 11-4. Simulator Command Summary (Continued)

Register Commands (continued)	
Command and Syntax	**Operation Description**
RR	Restore temporary copy of registers
SIO	Save temporary copy of I/O registers
SP *[32-bit value]*	Modify or display stack pointer
SR	Save temporary copy of registers
ST *[{ {N|C|V|I} {0|1} | 32-bit-value }]*	Modify or display the status register or specified status bit
Register Field Manipulation Commands	
Command and Syntax	**Operation Description**
CD *[{0|1}]*	Modify cache disable bit
CF *[{0|1}]*	Modify cache flush bit
FE0 *{0|1}*	Modify field extension of field 0
FE1 *{0|1}*	Modify field extension of field 1
FS0 *field-size*	Modify field size of field 0
FS1 *field-size*	Modify field size of field 1
IE *[{0|1}]*	Modify interrupt enable bit
ITPVH *[5-bit-value*	Modify or display ITPVH bits
NCZV *[4-bit-value*	Modify or display NCZV bits
PBH *[{0|1}]*	Modify or display PBH bit
PBV *[{0|1}]*	Modify or display PBV bit
PBX *[{0|1}]*	Set PBX status bit
PP *[pixel-processing-option]*	Modify or display pixel processing option
PS *[pixel-size]*	Set PSIZE register
T *[{0|1}]*	Toggle transparency bit
W *[{0|1|2|3}]*	Modify or display specified windowing option
Cache Manipulation Commands	
Command and Syntax	**Operation Description**
CD *[{0|1}]*	Modify cache disable bit
CF *[{0|1}]*	Modify cache flush bit
CLCS	Clear cache statistics
DC	Display cache contents and statistics
Breakpoint and Trace Commands	
Command and Syntax	**Operation Description**
BP	Display existing breakpoints
BP*n* *{Clear|OFf|ON|Toggle|Quit}*	Modify existing breakpoints
BPA{R|W|I|A} *{address|address-pattern}*	Set breakpoint on address
BPD{R|W|I|A} *{data|data-pattern*	Set breakpoint on data
BPR{R|W|I|A} *start-address end-address*	Set breakpoint on range
CTF	Close trace file
TR	Display existing traces
TR*n* *[{Clear|OFf|ON|Toggle|Quit}]*	Modify existing traces
TRA{R|W|I|A} *{address|address-pattern}*	Set trace on address
TRD{R|W|I|A} *{data|data-pattern}*	Set trace on data
TRR{R|W|I|A} *start-address end-address*	Set trace on range

Table 11-4. Simulator Command Summary (Concluded)

Host and Interrupt Control Commands	
Command and Syntax	**Operation Description**
I	Display current interrupts
In *[{ frequency-count | Clear|OFf|ON|Toggle|Quit }]*	Modify existing interrupts
IH *count frequency*	Initiate host input
MT *[OFF {27|28|29}]*	Modify special traps
CLS	Clear the graphics screen
G	Customize graphics environment
GR	Toggle graphics display
GR *{Clear|OFf|ON|Init|Disable|Enable}*	Execute graphics option
RGE *[file-number-extension]*	Restore graphics environment
RS	Regenerate graphics screen
SGE *[file-number-extension]*	Save graphics environment
TX	Toggle text display
Memory Environment Control Commands	
Command and Syntax	**Operation Description**
CLM	Clear simulator memory
DB *start-address [end-address]*	Display bytes
DM *start-address [end-address]*	Display memory
Dw *start-address [end-address]*	Display word of memory
F *start-address end-address 16-bit-value*	Fill memory with specified word
FW *[start-address end-address] 16-bit-value*	Find word
M	Customize memory environment
MM[+] *address {16-bit-value|32-bit-value| assembler-statement}*	*Modify or evaluate memory*
MMF[+] *address field-value field-size*	Modify memory field
Machine-State Environment Control Commands	
Command and Syntax	**Operation Description**
RMI *[file-number-extension [offset]]*	Restore memory image
RMS *[file-number-extension]*	Restore machine state
SMI *start-address end-address [file-number-extension]*	Save memory image
SMS *[file-number-extension]*	Save machine state
Debug Environment Control Commands	
Command and Syntax	**Operation Description**
DE	Toggle simulator debug mode
RDE *[file-number-extension]*	Restore debug environment
SDE *[file-number-extension]*	Save debug environment
Miscellaneous and Special Commands	
Command and Syntax	**Operation Description**
ID	Display simulator version identification
SF *filename*	Show file utility
SY *string*	Execute specified system function
V *value*	Evaluate data

Syntax **A**

Description The A command displays the A- and B-file registers. If the A- and B-file registers are already displayed, then the A command clears and rewrites the display, providing a full update of the register file contents.

Example Display the A- and B-file registers in the machine-state display.

```
Command[1] A CR
```

Figure 11-3 (page 11-5) shows a display that contains the general-purpose registers.

Syntax **An** *[32-bit-value]*

Description The A*n* command allows you to modify or display the contents of any of the A-file registers. This also allows you to view the contents of an A-file register when the text display is off.

n represents a number from 0–14; the default type for the register number is decimal. (To set or inspect the stack pointer, use the SP command.)

If you provide a *32-bit value*, it replaces the value of the specified A-file register. If you do not specify a replacement value, the simulator displays the current contents of the register. The default type for the value is hexadecimal.

Example 1 Modify the contents of register A3:

```
Command[1] A3 FFFFFFFE CR
```

Register A3 now contains the value 0FFFFFFFEh. Note that you could obtain the same result using the decimal type override, %-2.

Example 2 Display the contents of register A3:

```
Command[1] A3 CR
Command[1] A3 = FFFFFFFE
```

Now the command buffer shows the contents of register A3.

Note that this form of the command destroys any monitor commands that follow in the same buffer.

Syntax **B**

Description The B command causes the default register display to be the A- and B-file registers. If the A- and B-file registers are currently displayed, then the B command clears and rewrites the display, providing a full update of the register file.

Example Display the A- and B-file registers in the machine-state display:

```
Command[1] B CR
```

Figure 11-3 (page 11-5) shows a display that contains the general-purpose registers.

Syntax **B*n*** *[32-bit-value]*

Description The B*n* command allows you to modify or display the contents of any of the 15 B-file registers. This also allows you to view the contents of an A-file register when the text display is off.

n represents a number from 0–14. The default type for the register number is decimal. (To set or inspect the stack pointer, see the SP command.)

If you specify a *32-bit value,* it replaces the value of the specified B-file register. The default type for the value is hexadecimal. If you do not specify a replacement value, the simulator displays the current contents of the register.

Example 1 Modify the contents of register B3:

```
Command[1] B13 FFFFFFFF CR
```

Register B13 now contains the value 0FFFFFFFFh. Note that you could obtain the same result using the decimal type override, %-1.

Example 2 Display the contents of register B13:

```
Command[1]B13 CR
Command[1] B13 = FFFFFFFF
```

The contents of register B13 are now visible in the command buffer. Note that this form of the command destroys any monitor commands that follow in the same buffer.

> **Note:**
>
> The PIXBLT and FILL instructions use register B10–B14. When the TMS34010 executes one of these instructions, it does not save values the original values in these registers. Be careful that these instructions do not destroy data that you have stored in these registers.

Syntax **BP**

Description The BP command displays all existing breakpoints along with their active/inactive state. Figure 11-7 illustrates a display typically produced by this command.

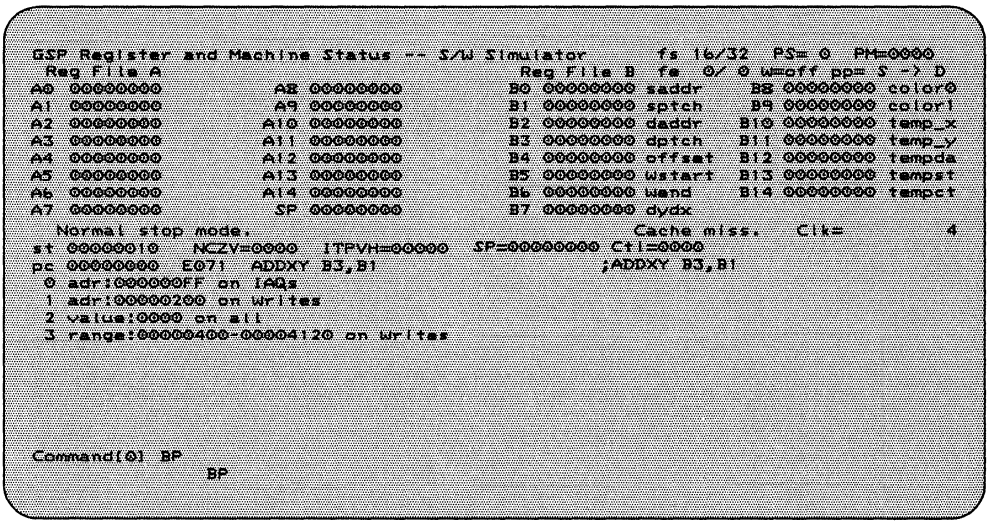


Figure 11-7. Display of Existing Breakpoints

Each breakpoint has a unique reference number (0 through 19). A combination of up to 20 breakpoints and traces can be defined. The reference numbers shown here are those used in conjunction with the BP*n* command to manipulate the state of each breakpoint on the list. If a breakpoint is inactive, it is displayed with a preceding asterisk.

Syntax **BP*n*** *[{ Clear | OFf | ON | Toggle | Quit }]*

Description The BP*n* command allows you to modify the status of individual breakpoints. A combination of up to 20 breakpoints and traces can be defined.

n is a breakpoint reference number (0–19) or the letter *X*. If you specify *X* as the breakpoint number, then **all** existing breakpoints are affected. The breakpoint reference number is displayed when the breakpoint is defined. The breakpoint reference number does not change throughout the life of the specific breakpoint. The BP command displays the breakpoint numbers.

Breakpoint options include:

?	Clear
?	destroys the breakpoint.
OFf	deactivates the breakpoint temporarily (but doesn't destroy it). An asterisk (*) next to a breakpoint number indicates that the breakpoint was deactivated.
ON	reactivates a breakpoint that has been turned off.
Toggle	activates an inactive breakpoint, or deactivates an active breakpoint.
Quit	terminates the command without changing any breakpoints.

Only the significant letters of each option (indicated by the uppercase letters in the list) are processed; CLEAR and C are treated the same.

If you do not enter the option on the command line, the simulator displays the breakpoint and a list of options for you to select from.

Example 1 Toggle breakpoint 3:

```
Command[1] BP3 T  CR
```

Example 2 Clear all breakpoints:

```
Command[1] BPX CLEAR   CR
```

Example 3 Enter a breakpoint command without an option:

```
Command[1] BP1 CR
```

The simulator displays the breakpoint and the following menu:

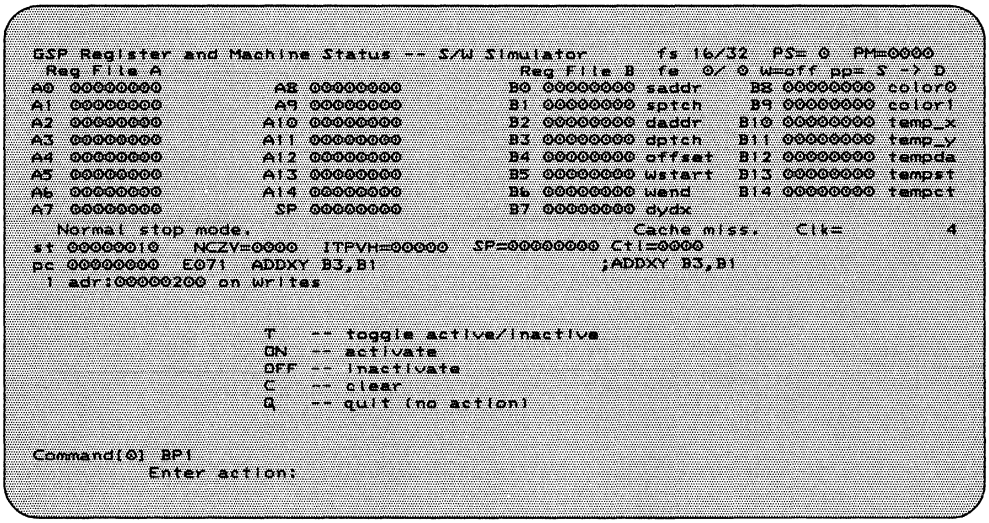


Figure 11-8. Modify Breakpoints Menu

Now you can enter any of the breakpoint options. For example, you could enter **T** to toggle the breakpoint. This toggles the state of breakpoint number 1 to off. Note that breakpoint 1 remains in memory and can be reactivated by the same command sequence or by specifying the ON option. Alternatively, it can be deleted with the CLEAR option and then overwritten by the BPA, BPD, or BPR command. You can verify the modification with the BP command.

Syntax

BPAR { *address* | *address-pattern* }

BPAW { *address* | *address-pattern* }

BPAI { *address* | *address-pattern* }

BPAA { *address* | *address-pattern* }

Description

The BPA command allows you to set breakpoints, stopping execution at:

- A specific *address* **or**
- An address that matches a specified *address pattern*.

There are four versions of this command:

BPAR breaks execution on all *memory reads* from the specified address.

BPAW breaks execution on all *memory writes* to the specified address.

BPAI breaks execution only on *instruction acquisitions* from the specified address.

BPAA breaks execution on *all memory accesses* to the specified address.

The default type for the address is hexadecimal. You can use a pattern instead of an address; a pattern is a 32-bit binary number with 1s and 0s in the data compare positions and Xs in the don't care positions. The pattern must be enclosed in parenthesis.

Example 1

Break execution when the simulator attempts to fetch an instruction from location 120FF310h.

```
Command[1] BPAI 120FF310 CR
```

Example 2

Break execution on any access to an address that matches the pattern X1XX0100XX00XXX0:

```
Command[1] BPAA (X1XX0100XX00XXX0) CR
```

Syntax

BPDR { *data* | *data-pattern* }

BPDW { *data* | *data-pattern* }

BPDI { *data* | *data-pattern* }

BPDA { *data* | *data-pattern* }

Description

This command allows you to set breakpoints, stopping execution when:

- A specific word of *data* is accessed **or**
- A word of data that matches a specified *data pattern* is accessed.

There are four versions of this command:

BPDR breaks execution on all *memory reads* of the specified data.

BPDW breaks execution on all *memory writes* to the specified data.

BPDI breaks execution only on *instruction acquisitions* of the specified data.

BPDA breaks execution on *all memory accesses* to the specified data.

The default type for the data is hexadecimal. You can use a pattern instead of a word of data; a pattern is a 16-bit binary number with 1s and 0s in the data compare positions and Xs in the don't care positions. The pattern is written within parentheses.

Example 1

Break execution on any data word access that matches 120Fh:

```
Command[1] BPDA 120F  CR
```

Example 2

Break execution any time a number matching the pattern is read from memory.

```
Command[1] BPDR (X1XX0100XX00XXX0) CR
```

Syntax

BPRR *start-address end-address*

BPRW *start-address end-address*

BPRI *start-address end-address*

BPRA *start-address end-address*

Description

This command allows you to set breakpoints, stopping execution on specified memory accesses within a range of addresses.

There are four versions of this command:

BPRR breaks execution on all *memory reads* of the specified data.

BPRW breaks execution on all *memory writes* to the specified data.

BPRI breaks execution only on *instruction acquisitions* of the specified data.

BPRA breaks execution on *all memory accesses* to the specified data.

The default type for the addresses is hexadecimal. You cannot specify a pattern for this command.

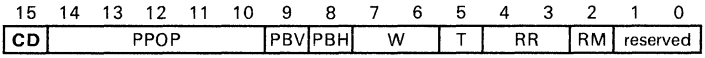
Example

Break execution on any access to locations 4000h through 4120h. The simulator issues a monitor error message if the start address is greater that the end address.

```
Command[1] BPRA 04000 4120 CR
```

Syntax **CD** *[{ 0 | 1 }]*

Description The CD command allows you to set, reset, or toggle the contents of the CD bit. CD is the cache disable bit (bit 15 in the CONTROL register):



CONTROL Register

The *0|1* parameter is optional; if you do not specify 0 or 1, the command toggles the current value of the CD bit. The *0|1* parameter has the following effects:

CD=0 enables the instruction cache.

CD=1 disables the instruction cache.

Note that the value of this bit is shown in the machine-state display.

Example Set the CD bit to 1 (this disables the cache):

```
Command[1] CD 1 CR
```

Syntax **CF** *[{ 0 | 1 }]*

Description The CF command allows you to set or reset the contents of the CF bit. CF is the cache flush bit (bit 14 in the HSTCTLH register):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HLT	**CF**	LBL	IN CR	IN CW	res	NM IM	NMI	reserved							

HSTCTLH Register

The *0|1* parameter is optional. If you do not specify 0 or 1, the command flushes the cache by setting all of the present flags to "not present" and sets the cache contents to all 0s (this does not affect the CF bit). The *0|1* parameter has the following effects:

CF=0 enables cache reads, depending upon the value of the cache disable (CD) bit.

CF=1 invalidates all current data in the cache. Cache accesses are inhibited until the CF bit is set to 0. The cache fragment present flags are also cleared.

Example Clear the CF bit in the HSTCTL register, enabling cache access:

```
Command[1] CF 0 CR
```

Syntax **CIF**

Description The CIF command closes the opened input file `gspinput.000`. You can restore execution of this file with the SWITCH command.

If a CIF command is encountered within a batch stream, then the input file is closed and execution continues from the beginning of the input file. (This produces a continuous loop.)

Example Close the input file:

```
Command[1] CIF CR
```

Syntax **CLA**

Description The CLA command clears (0s) all the A-file registers except the stack pointer. (Use the CLR or SP command to clear the stack pointer.)

Example Clear registers A0–A14 (SP is not changed).

```
Command[1] CLA CR
```

Syntax **CLB**

Description The CLB command clears (0s) all the B-file registers except the stack pointer. (Use the CLR or SP command to clear the stack pointer.)

Example Clear registers B0–B14 (SP is not changed).

```
Command[1] CLB CR
```

Syntax **CLCS**

Description The CLCS command clears the accumulated cache statistics and clears (0s) the cache present flags for all four cache segments. In addition, the actual cache contents are set to 0 (purged).

Example Clear the cache present flags, the cache statics, and the cache contents:

```
Command[1] CLCS CR
```

Syntax **CLIO**

Description The CLIO command clears (0s) all of the on-chip I/O registers. To inspect the I/O registers or to clear selected I/O registers only, use the IO command.

Example Clear all of the on-chip I/O registers:

```
Command[1] CLIO CR
```

Syntax **CLK** *[value]*

Description The CLK command allows you to display or modify the contents of the simulator clock. The CLK portion of the machine-state display normally shows the number of clock cycles consumed during instruction simulation. You can use the CLK command to view the clock contents when the machine-state display is toggled.

The *value* is optional; if specified, this becomes the new clock counter value. The default format for the clock counter is decimal.

Example 1 Set the clock count to 100:

```
Command [1] CLK 100  CR
```

Example 2 Display the clock count:

```
Command [1] CLK CR
Command [1] CLK = 100
```

Syntax **CLM**

Description The CLM command clears (0s) the entire range of simulator memory, from the end of simulator execution space to the end of writable memory.

> **Caution:**
>
> **CLM clears memory from the end of the simulator's execution space to the end of writable memory. If this memory is used for another purpose, such as a RAM disk, CLM destroys its contents.**

Example Clear the entire range of available memory:

```
Command[1] CLM CR
```

Syntax **CLR**

Description The CLR command clears:

- The A-file registers,
- The B-file registers, **and**
- The stack pointer.

To clear *only* the A-file registers, use the CLA command. To clear *only* the B-file registers, use the CLB command.

Example Clear all the general-purpose registers and the stack pointer:

```
Command[1] CLR CR
```

Syntax **CLS**

Description The CLS command clears (blanks):

- The simulated graphics from the host display surface **and**
- The scratch display area.

Example Clear the screen:

```
Command[1] CLS CR
```

Syntax **CTF**

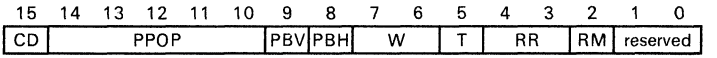
Description The CTF command closes the opened trace file named `gsptrace.000`. This allows you to use the SF (show file) command to inspect a file without exiting the simulator.

Example Close the trace file:

```
Command[1] CTF CR
```

Syntax **CTL** *[16-bit-value]*

Description The CTL command allows you to display or modify the contents of the CONTROL register.



CONTROL Register

The *16-bit value* is an optional parameter; its default type is hexadecimal. If you do not specify a replacement value, the simulator displays the CONTROL register's contents on the command line. This is useful for viewing the contents of the CONTROL register when the text display is off.

Example 1 Modify the contents of the CONTROL register:

```
Command[1] CTL 1046 CR
```

Example 2 Display the contents of the CTL register:

```
Command[1] CTL CR

Command[1] CTL 1046
             CTL 1046
```

The contents of the CONTROL register are now visible in the command buffer.

Note that this form of the command destroys any monitor commands that follow in the same buffer.

> **Note:**
>
> Bits 0 and 1 of the CONTROL register are reserved and cannot be modified. In these examples, the value in the CONTROL register after execution is 1044.

Syntax **DB** *start-address [end-address]*

Description The DB command displays blocks of TMS34010 memory. The *start address* and *end address* are expressed as 32-bit hexadecimal values.

Example Display a block of memory from address 0200h to 0550h:

```
Command[1] DB 200 550 CR
```

Although the address specified is a bit address, any bit address portion supplied is ignored, and the data is specified in words. The resulting display is shown below in the default display mode.

```
GSP Register and Machine Status -- S/W Simulator      fs 16/32  PS= 4  PM=0000
  Reg File A                                Reg File B  fe  0/ 0 W=off pp= S -> D
A0 00000000          A8 00000000          B0 00000000 saddr    B8 00000000 color0
A1 00000000          A9 00000000          B1 00000000 sptch    B9 00000000 color1
A2 00000000          A10 00000000         B2 00000000 daddr    B10 00000000 temp_x
A3 00000000          A11 00000000         B3 00000000 dptch    B11 00000000 temp_y
A4 00000000          A12 00000000         B4 00000000 offset   B12 00000000 tempda
A5 00000000          A13 00000000         B5 00000000 wstart   B13 00000000 tempst
A6 00000000          A14 00000000         B6 00000000 wend     B14 00000000 tempct
A7 00000000           SP 00000000         B7 00000000 dydx
  Normal stop mode.                                  Cache miss.   Clk=          4
st 00000010   NCZV=0000  ITPVH=00001  SP=00000000 Ctl=0110
pc 00000000  0002  INVALID OP                      ;INVALID OP
  Address    LSB   Memory in bytes in ascending order.   MSB ASCII Characters
  00000200   40 00 40 00 40 00 40 00 40 00 40 00 40 00 40 00 @.@.@.@.@.@.@.@.
  00000280   40 00 40 00 40 00 40 00 40 00 40 00 40 00 40 00 @.@.@.@.@.@.@.@.
  00000300   40 00 40 00 40 00 40 00 40 00 40 00 40 00 40 00 @.@.@.@.@.@.@.@.
  00000380   40 00 40 00 40 00 40 00 40 00 40 00 40 00 40 00 @.@.@.@.@.@.@.@.
  00000400   40 00 40 00 40 00 40 00 40 00 40 00 40 00 40 00 @.@.@.@.@.@.@.@.
  00000480   40 00 40 00 40 00 40 00 40 00 40 00 40 00 40 00 @.@.@.@.@.@.@.@.
  00000500   40 00 40 00 40 00 40 00 40 00 40 00 02 00 02 00 @.@.@.@.@.@.....


Command[0]DB 200 550
          DB 200 550
```

Figure 11-9. Display Bytes Format

The simulator can display only 10 lines of information per screen. If you request more than 10 lines, the simulator halts the display; enter a carriage return to continue the display (see Figure 11-10). Enter a Q to quit the display.

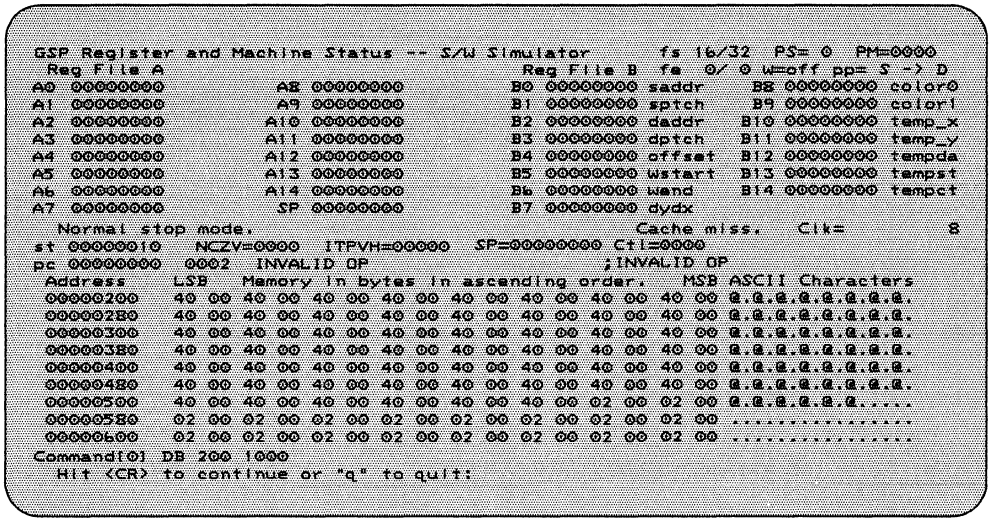


Figure 11-10. Display Bytes Format – Over 10 Lines

Note that the start address must be less than the end address for the DB command to operate, or the simulator issues an error message indicating that the start of the range exceeds the end.

Syntax **DC**

Description The DC command displays the cache contents and statistics on the screen. The statistical information is collected only when the cache is enabled.

Each cache page is displayed in the order of the LRU (least recently used) queue. The queue number in the display header refers to the current queue number. Queue numbers are displayed in ascending order from 0 to 3. The page number in the display refers to the actual cache page number at the currently displayed position within the LRU queue. Within each page, cache fragments are displayed as four consecutive words. As each instruction is encountered, it is disassembled (this is shown on the right side of the display). Valid cache fragments that have been loaded by executing code are displayed in cyan with yellow stars on the left side.

Example Display the cache contents and statistics in the scratch-display area:

```
Command[1] DC CR
```

```
GSP Register and Machine Status -- S/W Simulator      fs 16/32  PS= 0  PM=0000
  Reg File A                                Reg File B  fe  0/ 0 W=off pp= S -> D
A0 00000000          A8 00000000          B0 00000000 saddr     B8 00000000 color0
A1 00000000          A9 00000000          B1 00000000 sptch     B9 00000000 color1
A2 00000000         A10 00000000          B2 00000000 daddr    B10 00000000 temp_x
A3 00000000         A11 00000000          B3 00000000 dptch    B11 00000000 temp_y
A4 00000000         A12 00000000          B4 00000000 offset   B12 00000000 tempda
A5 00000000         A13 00000000          B5 00000000 wstart   B13 00000000 tempst
A6 00000000         A14 00000000          B6 00000000 wend     B14 00000000 tempct
A7 00000000          SP 00000000          B7 00000000 dydx
   Normal stop mode.                                  Cache miss.    Clk=          8
st 00000010   NCZV=0000  ITPVH=00000  SP=00000000 Ctl=0000
pc 00000000  0002  INVALID OP                    ;INVALID OP
             Cache statistics
        Cache hits     =          0.
   Total Cache accesses =          0.
         No statistics yet generated.

   Cache is displayed by page beginning with
         the MOST recently used page.



 Command[0] DC
 Hit <CR> to continue or "q" to quit:
```

```
GSP Register and Machine Status -- S/W Simulator      fs 16/32  PS= 0  PM=0000
  Reg File A                                  Reg File B  fe  0/ 0 w=off pp= S -> D
A0 00000000          A8 00000000          B0 00000000 saddr     B8 00000000 color0
A1 00000000          A9 00000000          B1 00000000 sptch     B9 00000000 color1
A2 00000000         A10 00000000          B2 00000000 daddr    B10 00000000 temp_x
A3 00000000         A11 00000000          B3 00000000 dptch    B11 00000000 temp_y
A4 00000000         A12 00000000          B4 00000000 offset   B12 00000000 tempda
A5 00000000         A13 00000000          B5 00000000 wstart   B13 00000000 tempst
A6 00000000         A14 00000000          B6 00000000 wend     B14 00000000 tempct
A7 00000000          SP 00000000          B7 00000000 dydx
  Normal stop mode.                                  Cache miss.   Clk=          8
st 00000010   NCZV=0000  ITPVH=00000  SP=00000000 Ctl=0000
pc 00000000  0002  INVALID OP                        ;INVALID OP
  Cache addr    Data  Rev-assembly   Queue no. 0  Page no. 3 CD= 0
    00000000 ==> 0000  INVALID OP
    00000010 ==> 0000  INVALID OP
    00000020 ==> 0000  INVALID OP
    00000030 ==> 0000  INVALID OP

    00000040 ==> 0000  INVALID OP
    00000050 ==> 0000  INVALID OP
    00000060 ==> 0000  INVALID OP
    00000070 ==> 0000  INVALID OP
Command[0] DC
Hit <CR> to continue or "q" to quit:
```

Figure 11-11. Cache Contents Display

Syntax **DM** *start-address [end-address]*

Description The DM command displays blocks of TMS34010 memory. The *start address* and *end address* are expressed as 32-bit hexadecimal values.

Example Display the block of memory between addresses 0200h–0550h.

```
Command[1] DM 200 550 CR
```

Although the address specified is a bit address, any bit address portion supplied is ignored, and the data is specified in words. The resulting display is shown below in the default display mode.

```
GSP Register and Machine Status -- S/W Simulator      fs 16/32  PS= 0  PM=0000
  Reg File A                                  Reg File B  fe  0/ 0 W=off pp= S -> D
A0 00000000          A8 00000000          B0 00000000 saddr    B8 00000000 color0
A1 00000000          A9 00000000          B1 00000000 spitch   B9 00000000 color1
A2 00000000          A10 00000000         B2 00000000 daddr    B10 00000000 temp_x
A3 00000000          A11 00000000         B3 00000000 dpitch   B11 00000000 temp_y
A4 00000000          A12 00000000         B4 00000000 offset   B12 00000000 tempdа
A5 00000000          A13 00000000         B5 00000000 wstart   B13 00000000 tempst
A6 00000000          A14 00000000         B6 00000000 wend     B14 00000000 tempct
A7 00000000           SP 00000000         B7 00000000 dydx
  Normal stop mode.                                  Cache miss.   Clk=          8
st 00000010   NCZV=0000  ITPVH=00000  SP=00000000 Ctl=0000
pc 00000000   0002  INVALID OP                    ;INVALID OP
  Addr (lsb)  msb  (== Memory in ascending bit order ==  lsb  ASCII Characters
  00000200   0040  0040  0040  0040  0040  0040  0040  0040  @.@.@.@.@.@.@.@.
  000002B0   0040  0040  0040  0040  0040  0040  0040  0040  @.@.@.@.@.@.@.@.
  00000300   0040  0040  0040  0040  0040  0040  0040  0040  @.@.@.@.@.@.@.@.
  000003B0   0040  0040  0040  0040  0040  0040  0040  0040  @.@.@.@.@.@.@.@.
  00000400   0040  0040  0040  0040  0040  0040  0040  0040  @.@.@.@.@.@.@.@.
  000004B0   0040  0040  0040  0040  0040  0040  0040  0040  @.@.@.@.@.@.@.@.
  00000500   0002  0002  0040  0040  0040  0040  0040  0040  @.@.@.@.@.@.....


Command[0] DM 200 550
           DM 200 550
```

Figure 11-12. Memory Display

The simulator can display only 10 lines of information per screen. If you request more than 10 lines, the simulator halts the display; enter a carriage return to continue the display. Enter a Q to quit the display.

Note that the start address must be less than the end address for the DM command to operate, or the simulator issues an error message.

Syntax **DR**

Description The DR command toggles the machine-state display between the I/O registers and the general-purpose registers.

Example 1 Display the A- and B-file registers in the machine-state display:

```
Command[1] DR CR
```

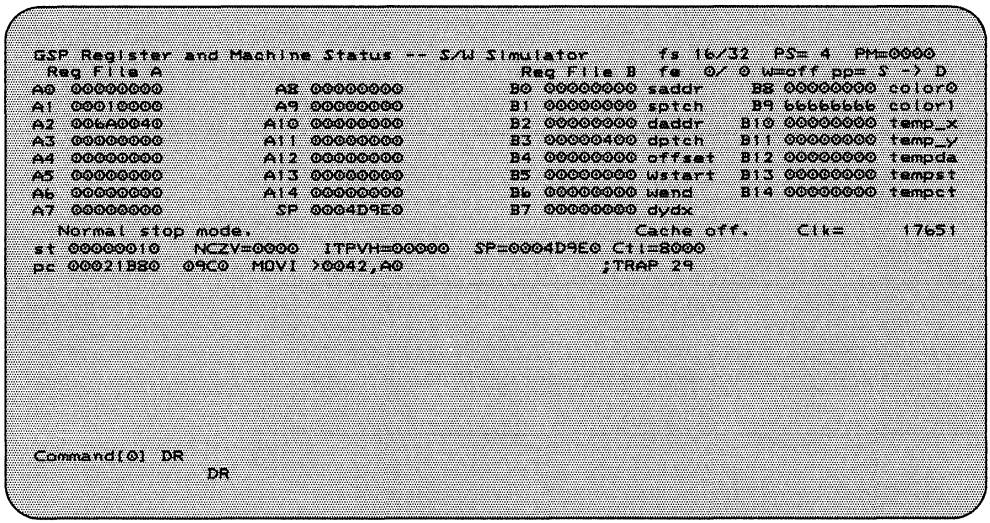


Figure 11-13. A- and B-File Registers Display

Example 2 Now display the I/O registers in the machine-state display:

```
Command[1] DR CR
```

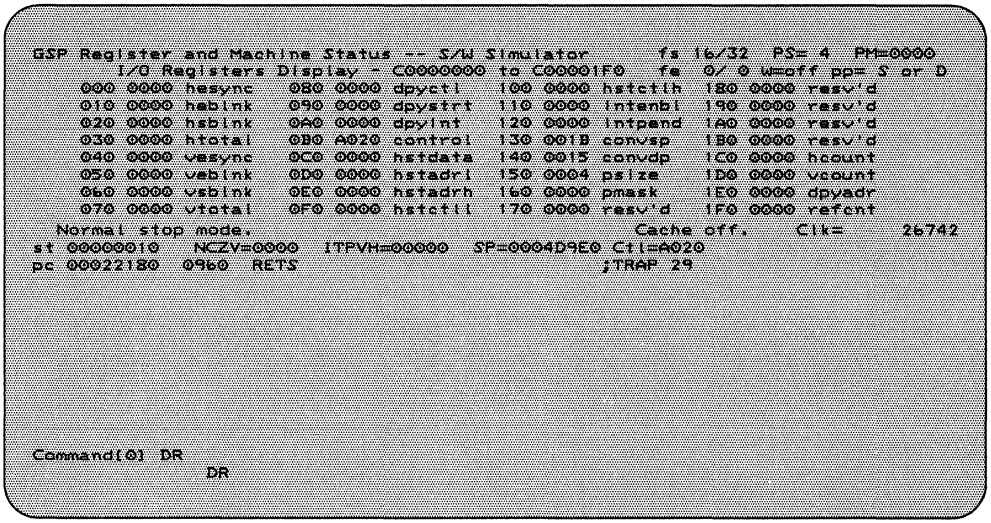


Figure 11-14. I/O Registers Display

Syntax **Dw** *start-address [end-address]*

Description The Dw command displays blocks of TMS34010 memory. The *start address* and *end address* are expressed as 32-bit hexadecimal values.

Example Display a block of memory from address 0200h to 0500h:

```
Command[1] D 00200 550  CR
```

or

```
Command[1] DW 00200 550  CR
```

Although the addresses are specified as bit addresses, any bit address portion supplied is ignored, and the data is specified in words. The resulting display is shown below in the default display mode.

```
GSP Register and Machine Status -- S/W Simulator      fs 16/32  PS= 0  PM=0000
 Reg File A                                  Reg File B  fe  0/ 0 w=off pp= S -> D
A0 00000000          A8 00000000          B0 00000000 saddr    B8 00000000 color0
A1 00000000          A9 00000000          B1 00000000 sptch    B9 00000000 color1
A2 00000000          A10 00000000         B2 00000000 daddr    B10 00000000 temp_x
A3 00000000          A11 00000000         B3 00000000 dptch    B11 00000000 temp_y
A4 00000000          A12 00000000         B4 00000000 offset   B12 00000000 tempda
A5 00000000          A13 00000000         B5 00000000 wstart   B13 00000000 tempst
A6 00000000          A14 00000000         B6 00000000 wend     B14 00000000 tempct
A7 00000000          SP 00000000          B7 00000000 dydx
  Normal stop mode.                                  Cache miss.   Clk=          8
st 0000010  NCZV=0000  ITPVH=00000  SP=00000000 Ctl=0000
pc 00000000  0002  INVALID OP                    ;INVALID OP
 Address    LSW   Memory in Words in ascending order.  MSW   ASCII Characters
 00000200   0040  0040  0040  0040  0040  0040  0040  0040  @.@.@.@.@.@.@.@.
 00000280   0040  0040  0040  0040  0040  0040  0040  0040  @.@.@.@.@.@.@.@.
 00000300   0040  0040  0040  0040  0040  0040  0040  0040  @.@.@.@.@.@.@.@.
 00000380   0040  0040  0040  0040  0040  0040  0040  0040  @.@.@.@.@.@.@.@.
 00000400   0040  0040  0040  0040  0040  0040  0040  0040  @.@.@.@.@.@.@.@.
 00000480   0040  0040  0040  0040  0040  0040  0040  0040  @.@.@.@.@.@.@.@.
 00000500   0040  0040  0040  0040  0040  0040  0002  0002  @.@.@.@.@.@.....


Command[0] D 200 550
           D 200 550
```

Figure 11-15. Display Word Format

The simulator can display only 10 lines of information per screen. If you request more than 10 lines, the simulator halts the display; enter a carriage return to continue the display. Enter a Q to quit the display.

Note that the start address must be less than the end address for the D command to operate, or the command line returns with no action.

Syntax **F** *start-address end-address 16-bit-value*

Description The F command fills a block of memory from the start address to the end address with the specified word value. You can use F to fill screen memory, program memory, or both; the simulator does not distinguish between the two.

The *start address* must be less than the *end address*. The default type for both addresses is hexadecimal. The bit address of both addresses is truncated to form a word address.

The default type for the *16-bit value* is hexadecimal; this parameter specifies a fill value.

You can use the F command to fill screen memory, program memory, or both; the simulator does not distinguish between the two.

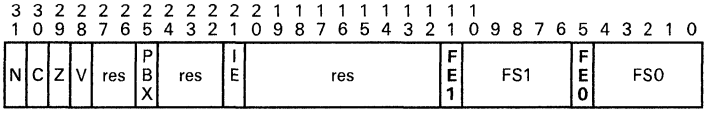
Example Fill memory from 200h to 350h, inclusive, with 00AAh:

```
Command[1] F 200 350 AA  CR
             F 200 350 AA
```

Syntax **FE0** { *0* | *1* }

FE1 { *0* | *1* }

Description The FE command sets the values of the FE0 or FE1 bits to select zero-extension or sign-extension for field 0 or field 1. The FE bits are the field extension bits (bit 5 and bit 11 the status register):



Status Register

The *0|1* parameter has the following effects:

FE0=0 selects zero extension for field 0.
FE0=1 selects sign extension for field 0.

FE1=0 selects zero extension for field 1.
FE1=1 selects sign extension for field 1.

The **FE** field in the machine-state display reflects the modifications you make with the FE command.

Example 1 Set FE0 to 1, enabling sign extension for field 0:

```
Command[1] FE 0 1 CR
```

or

```
Command[1] FE0 1 CR
```

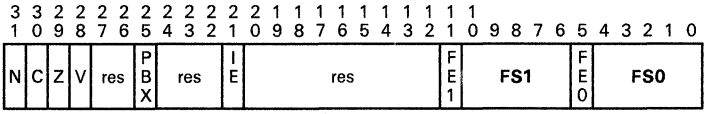
Example 2 Modify the contents of the FE1 bit to enable zero extension for field 1:

```
Command[1] FE1 0 CR
```

Syntax **FS0** *field-size*

FS1 *field-size*

Description The FS command sets the values of the FS0 or FS1 bits to select a new field size for field 0 or field 1. The FS bits are the field size bits (bits 0–4 and 6–10 in the status register):



Status Register

The default type for the field size is a decimal value from 1 to 32. The **FS** field in the machine-state display reflects any modifications you make with the FS command.

Example 1 Change the field size of field 0 to 16:

```
Command[1] FS0 16 CR
```

Example 2 Change the field size of field 1 to 4:

```
Command[1] FS1 4 CR
```

Syntax **FW** *[start-address end-address] 16-bit-value*

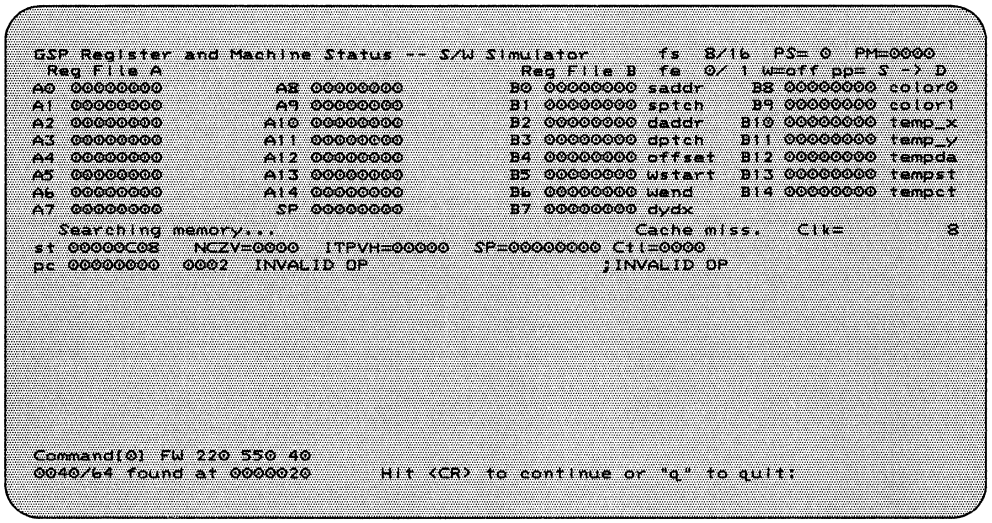
Description The FW command finds a specified 16-bit value within a range of memory. The simulator displays the value in hexadecimal and decimal, along with the address at which it was found. If no value is displayed, then the value was not found.

The default format for the value is hexadecimal. The addresses are specified as 32-bit addresses; the default format for the addresses is hexadecimal. The addresses must be word aligned; if they are not, then they are forced to be word aligned by setting the four LSBs to 0.

Example Find the value 40h in the range 220h–550h:

```
Command[1] FW 220 550 40 CR
```

Figure 11-16 shows an example of this command that looks for the word value 40 within a defined address range.



Syntax **G**

Description The G command initializes or modifies the graphics definitions for the graphics environment simulation, including screen limits and the pixel size. You can use the SGE command to store this data in a file.

Example Enter the graphics environment customization utility:

```
Command[1] G CR
```

The simulator displays the screen shown in Figure 11-17.

```
                Graphics Environment Customization Commands
M -- set Memory Begin and End              D -- set H/W Window's Dimensions
B -- set Screen Memory Begin               G -- set Window's GSP position
P -- set Screen Memory Pitch               H -- set HOst Display Dimension
F -- set Display's First Pixel Address     S -- set Window's Monitor Position
X -- set H/W Pixel Size                    R -- Regenerate Screen
                                           Q -- Quit Customization
                      ---- GSP Memory Definitions ----
 General Memory: Begin     End          Screen Memory: Begin      Pitch (pixels)
                 00000000  0041C000                   00000000           256

                      ---- GSP Screen Display Area ----
          First Displayed Pixel          Pixel Size     Display Dimensions
                -X-          -Y-             4            Width   Height
                 0            0                           256     128

                      ---- Host Window Area ----
        GSP POsition       Host Dimensions        Monitor Screen Position
         -X-    -Y-         Width   Height             -X-    -Y-
          0      0           720     276               386    170
 Choice:
```

Figure 11-17. Graphics Customization Menu

Syntax **GR**

Description The GR command and the TX command select between the graphics and nongraphics displays. GR toggles the graphics so that the text display can be clearly seen.

On systems with independent graphics display, such as the TI-PC, GR does not affect the graphics display. On systems with integrated text and graphics, such as the IBM-PC, both the GR and TX commands toggle between text and graphics.

TX toggles the text display in the same way that GR toggles the graphics display. When the text display is off, the command line remains in its usual position, but the simulator status line replaces the previous command entry line.

Example Turn the graphics display on: `Command[1] GR` ***CR***

```
Command[1] GR
  Normal Stop Mode.
```

Note that monitor status messages now appear below the command line:

```
Command[1] RUN
    Simulator Running ...
```

You can use the same command to toggle the graphics display back off.

Syntax **GR**{ *CLeaR* | *OFF* | *ON* | *INIT* | *DISable* | *ENaBle* }

Description The GR command provides the following options:

GRCCLR *clears* the graphics screen of interpreted graphics memory. This allows you to clear the screen of graphics display so that only newly written graphics data appears on the screen. This does not affect the text display.

GROFF Turn the graphics system off. While the display is off, the graphics memory is still being written to, so that when the display is restored, any effect on graphics memory is restored to the screen. This command is also useful to view text such as status information clearly when the screen is full of graphics information.

GRON turns the graphics display *on*

GRINIT *initializes* the graphics system. This returns the graphics environment to its initial state at invocation, thus returning it to a known state.

GRDIS *disables* any additional host graphics simulation. This allows the simulation to bypass the overhead of interpreting the impact of memory accesses on the graphics display, producing a faster functional simulation while the utility of regenerating the screen using the RS command remains. However, you do not losed the full, concurrent graphics simulation.

GRENB re-*enables* the host graphics simulation after it it is disabled. This allows additional graphics displays to be routed to the screen.

The following examples illustrate these functions.

Example 1 Clear the graphics display.

```
Command[1] GRCLR CR
```

or

```
Command[1] GR CLR CR
```

Although the graphics display is cleared, its on/off status is not changed.

Example 2 Disable host graphics simulation.

```
Command[1] GRDIS CR
```

or

```
Command[1] GR DIS CR
```

Although the graphics display is disabled, its on/off status is not changed and data displayed on the screen remains. You can restore the display with the RS (regenerate screen) command.

Example 3 Turn the graphics display on:

```
Command[1] GRON CR
             A3 464

Command[1] GRON
             GRON
```

Example 4 Turn the graphics display off.

```
Command[1] GR CR
             A3 464

Command[1] GR
             GR
```

Syntax **Help**

Description The HELP command displays a menu of help files.

When you select a help file, the simulator displays the file in the same manner as the SF (show file) command. The simulator informs you if a help file is not available.

Example Display the help menu:

`Command[1]` **HELP** ***CR***

The simulator enters the help environment and displays a menu that describes the utility and lists help files for the various classes of commands.

```
                    GSP SIMULATOR HELP FUNCTION

    B -- help breakpoint/trace commands        P -- help program execution
                                                     commands
    E -- help for environment
           retention commands                  R -- help register/status
                                                    display/modify commands
    G -- help graphics commands
                                               S -- help simulator specific
    I -- help interrupt/host interface             commands
           commands
                                               O -- simulator overview
    M -- help memory manipulation commands

    Q -- Quit help

  Choices:
```

Figure 11-18. Simulator Help Utility Menu

Syntax I

Description The I command displays all currently defined interrupts, including their active/inactive status. Figure 11-19 shows a typical display produced by the I command.

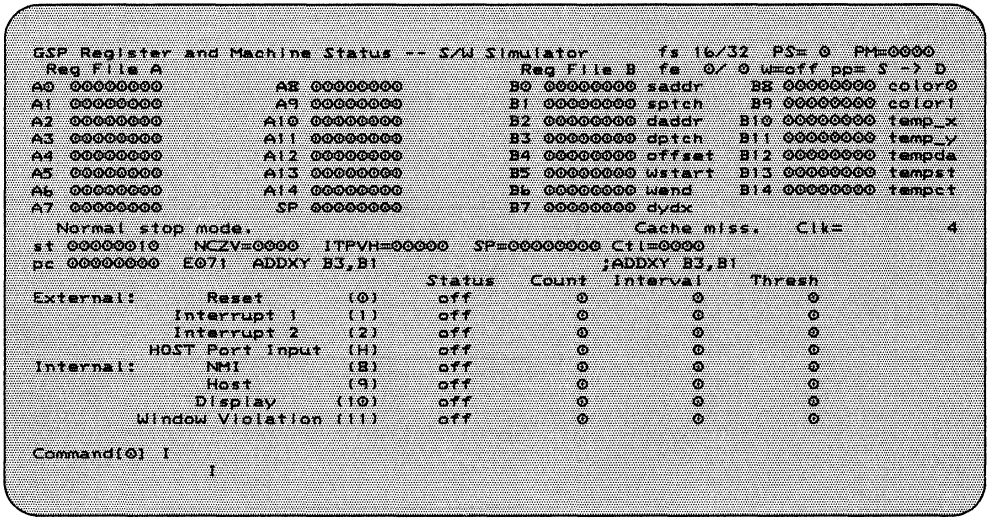


Figure 11-19. Interrupt Displays

Note that each interrupt is referenced by its level number (0–2, H, 8–11) and that any combination of the TMS34010 interrupts can be active at any specified time. Interrupt level H is assigned for simulation purposes in the host input.

Syntax **I*n*** *[{ frequency-count | Clear | OFf | ON | Set | Toggle | Quit }]*

Description The I*n* command allows you to set the status of individual interrupts. (Use the I command to display interrupt status.)

> **Notes:**
>
> 1. If you request a nonexistent interrupt, the command is invalid and the simulator issues an error message.
> 2. Interrupt priority is preserved during simulation.

Executing the I command a second time for the same interrupt overwrites the previous information for that interrupt. You can also clear, turn off, turn on, toggle the interrupt, or quit the command execution.

The interrupt number (*n*) corresponds to the number which is specified in the I command display. This number is a decimal number 0–2 or 8–11 or H. Specifying *X* as the interrupt number makes the I command apply to all the interrupts. Specifying *H* as the interrupt number refers to host input interrupt.

Example 1 Generate interrupt $\overline{\text{INT}}$2 every 0F0h clock cycles for a total of three times:

```
Command[1] I 2 S F0H 3   CR
```

or

```
Command[1] I2 S F0H 3   CR
```

Example 2 Clear all interrupts:

```
Command[1] IX C   CR
```

Example 3 Toggle interrupt 3:

```
Command[1] I3 T    CR
```

or

```
Command[1] I3 CR
```

The last entry displays the interrupt and the following menu:

```
GSP Register and Machine Status -- S/W Simulator      fs 16/32  PS= 0  PM=0000
 Reg File A                                Reg File B  fe  0/ 0 W=off pp= S -> D
A0 00000000          A8 00000000          B0 00000000 saddr     B8 00000000 color0
A1 00000000          A9 00000000          B1 00000000 sptch     B9 00000000 color1
A2 00000000          A10 00000000         B2 00000000 daddr     B10 00000000 temp_x
A3 00000000          A11 00000000         B3 00000000 dptch     B11 00000000 temp_y
A4 00000000          A12 00000000         B4 00000000 offset    B12 00000000 tempda
A5 00000000          A13 00000000         B5 00000000 wstart    B13 00000000 tempst
A6 00000000          A14 00000000         B6 00000000 wend      B14 00000000 tempct
A7 00000000          SP 00000000          B7 00000000 dydx
  Normal stop mode.                                  Cache miss.    Clk=          4
st 00000010   NCZV=0000  ITPVH=00000  SP=00000000 Ctl=0000
pc 00000000  E071  ADDXY B3,B1                    ;ADDXY B3,B1


                   T   -- toggle active/inactive
                   ON  -- activate
                   OFF -- inactivate
                   C   -- clear
                   S   -- set threshold and count
                   Q   -- quit (no action)


Command[0] IO
         Enter action:
```

Figure 11-20. Display Interrupt Options

At this point, you can enter T. This toggles the state of interrupt number 2. Note that interrupt 2 remains in memory; you can reactivate it with the same command sequence. Alternatively, you can use this same command to clear or reinitiate the interrupt. The command entry is verified by the display of interrupts exactly as it appears after an I command.

Syntax **ID**

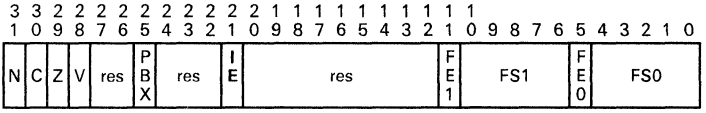
Description The ID command prints the version of the simulator below the command line.

Example Display the simulator version number:

```
Command[1]  ID  CR
                Version 1.04120886
```

Syntax **IE** *[{ 0 | 1 }]*

Description The IE command allows you to set, reset, or toggle the contents of the IE (interrupt enable) bit in the status register.



Status Register

The *0|1* parameter has the following effects:

IE=0 disables all maskable interrupts.

IE=1 enables all maskable interrupts.

If you do not use the *0|1* parameter, the IE bit is toggled. Note that the value of this bit is shown in the **I** portion of the machine state's ITPVH field.

Example Set the IE bit:

```
Command[1] IE 1 CR
```

This sets the IE bit, which disables interrupts.

Syntax **IH** *count frequency*

Description The IH command is used in conjunction with a host input file (`host.in`) to simulate the interface between the TMS34010 and a host. You can specify how often a host input will occur; *count* number of host inputs are generated every *frequency* clock cycles. The simulator generates *count* host inputs every *frequency* clock cycles. If *frequency* is 0, no host input "interrupts" are allowed; in effect, you are turning host input mode off. By default, host inputs are disabled.

When the first host input is generated, the simulator reads the last status information from `host.in`. The information contained in this file represents the pin and data bus values the host would ordinarily generate. Also contained in this file is the number of times that each kind of input is to occur. Thus, subsequent host input "interrupts" only access the host input file when the previously read action information has been completed. `host.in` contains one host input type per line and its format.

Note that if the host wants to read from the TMS34010, then the value read (contained on the HD bus) is written into the host output file, `host.out`. `host.out` will contain one 16-bit word per line (hexadecimal format), along with the clock value at the point at which it occurs.

Example Generate a host input 15 times every 150 clock cycles; note that both the clock count and frequency are decimal numbers.

```
Command[1] IH 150 15   CR
```

Syntax **IO**

Description The IO command explicitly sets the default registers in the status display to the I/O register. You can use the DR, A, or B commands to to change the display between the I/O registers and the A- and B-file registers.

Example Display the I/O registers in the machine-state display:

```
Command[1] IO CR
```

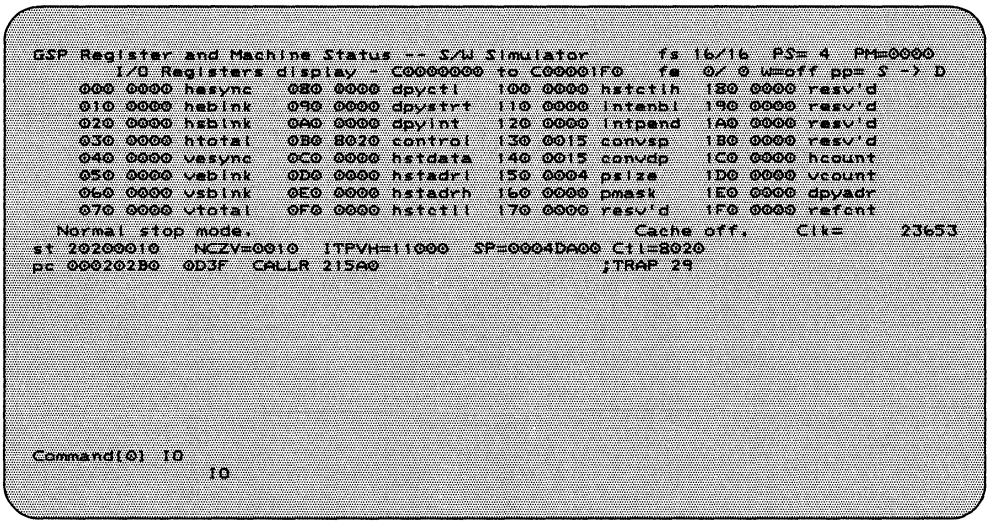


Figure 11-21. I/O Registers Display

Syntax **IO *n*** [*value*]

Description The IO*n* command allows you to modify the contents of any of the memory-mapped I/O registers simply by specifying the offset from the I/O register's base address (0C0000000h). You can also use this command to display the contents of a particular I/O register; to do this, use the reference number without specifying a replacement value.

Table 11-5 lists the I/O registers and their offsets.

Table 11-5. I/O Registers and Offsets

Offset†	Register	Description	Offset†	Register	Description
000	HESYNC	Horizontal end sync	100	HSTCTLH	Host control high
010	HEBLNK	Horizontal end blank	110	INTENB	Interrupt enable
020	HSBLNK	Horizontal start blank	120	INTPEND	Interrupt pending
030	HTOTAL	Horizontal end total	130	CONVSP	Source pitch
040	VESYNC	Vertical end sync	140	CONVDP	Destination pitch
050	VEBLNK	Vertical end blank	150	PSIZE	Pixel size
060	VSBLNK	Vertical start blank	160	PMASK	Plane mask
070	VTOTAL	Vertical total	170	–	Reserved
080	DPYCTL	Display control	180	–	Reserved
090	DPYSTRT	Display start	190	–	Reserved
0A0	DPYINT	Display interrupt	1A0	–	Reserved
0B0	CONTROL	Control	1B0	DPYTAP	Display tap address
0C0	HSTDATA	Host data	1C0	HCOUNT	Horizontal count
0D0	HSTADRL	Host address low	1D0	VCOUNT	Vertical count
0E0	HSTADRH	Host address high	1E0	DPYADR	Display address
0F0	HSTCTLL	Host control low	1F0	REFCNT	DRAM refresh count

† The offset is added to a base address of 0C0000000h.

Example 1 Modify the contents of the I/O register located at address 0C00000F0h:

```
Command[1] IO F0 F046   CR
```

The HSTCTLL register now contains 0F046h.

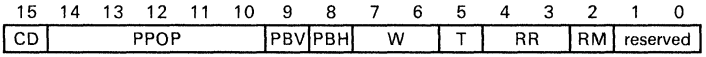
Example 2 Inspect the contents of the HSTCTLL register:

```
Command[1] IO F0 CR
Command[1] IO F0 = F046
```

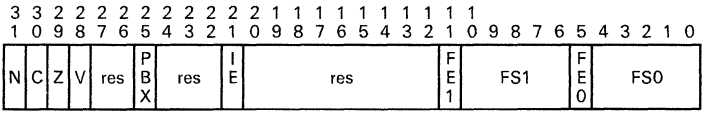
Syntax **ITPVH** *[5-bit-value]*

Description The ITPVH command allows you to display or modify the contents of the ITPVH bits, found in the status and CONTROL registers:

I Interrupt enable bit (status bit 21)
T Transparency bit (CONTROL bit 5)
P PixBlt executing bit (status bit 25)
V PixBlt vertical bit (CONTROL bit 9)
H PixBlt horizontal bit (CONTROL bit 8)



CONTROL Register



Status Register

To modify any of the bits, enter a *5-bit value* composed of 0s and 1s. A value of 1 sets the bit; a value of 0 clears the bit.

To display the current contents of these bits, enter the command without any value.

Example 1 Reset the ITPVH bits to 0s:

```
Command [0] ITPVH 00000 CR
```

Example 2 Display the values of the ITPVH bits:

```
Command [0] ITPVH CR
Command [0] ITPVH = 00000
```

Example 3 Enable interrupts and turn transparency on:

```
Command [0] ITPVH 11000  CR
```

Syntax **L** *filename [offset]*

Description The L command loads a linked, executable COFF module into the simulator local memory so that it can be executed. You can use the optional *offset* parameter to relocate the module at load time. Note that the simulator cache is flushed on a successful load so that old code in cache is not executed.

The *filename* names the module that is loaded into memory. You can specify a file without an extension; ***.out*** is the default extension for modules produced by the linker and ***.obj*** is the default extension for modules produced by the assembler. If you do not specify an extension for *filename*, the simulator first attempts to load `filename`. If this file isn't found, the simulator attempts to load `filename.out`. If this file isn't found, the simulator attempts to load `filename.obj`. You *can* load unlinked code, but the simulator issues a warning message, and unresolved references are not resolved.

If you specify an *offset*, then the simulator attempts to relocate the module when loading by adding the offset to all relocation entries in the module. The *offset* is treated as a signed 32-bit quantity. If you attempt to load an absolute (nonrelocatable) module, the simulator issues a warning and ignores the *offset*. If you do not specify an *offset*, then all relocation entries are loaded relative to 0.

The graphics display is inhibited to speed the load. Data loaded into defined screen memory does not appear on the graphics display. You can use the RS command to see the graphic interpretation of the loaded data.

Example 1 Load module `code.out` with offsets 0 and 1000D00h, respectively:

```
Command[1] L CODE CR

Command[1] L CODE.OUT 1000D00 CR
```

The second example above causes the simulator to open and read the file `code.out`, interpret it, and load it into the simulated TMS34010 memory with an offset of 1000D00h.

Example 2 Load a COFF file from the directory `\LASER\OUT` on drive C:

```
Command[0] L C:\LASER\OUT\CODE CR
```

Syntax **LE**

Description The LE command displays the most recent set of error messages after they are removed from the screen. Error messages appear in red in the scratch-display.

Example Display previous error messages:

```
Command[1] LE CR
```

Syntax **LH**

Description The LH command displays the most recent set of halt messages after they are removed from the screen. Halt messages, which are generated by encountering breakpoints, appear in cyan in the scratch-display.

Example Display previous halt messages:

```
Command[1] LH CR
```

Syntax **LM**

Description The LM command displays the most recent set of monitor error messages after they are removed from the screen. Monitor error messages appear in yellow in the monitor-message display.

Example Display previous monitor messages:

```
Command[1] LM CR
```

Syntax **M**

Description The M command initializes or modifies the memory definitions for the simulation. The memory values that can be modified are the general and the screen memory limits. You can use the SGE command to store this data along with the graphics environment information.

Example `Command[1] M` ***CR***

The simulator displays the menu shown in Figure 11-22. This menu allows you to selectively modify the memory environment.

```
                    Graphics Environment Customization Commands
M -- set Memory Begin and End              D -- set H/W Window's Dimensions
B -- set Screen Memory Begin               G -- set Window's GSP position
P -- set Screen Memory Pitch               H -- set HOst Display Dimension
F -- set Display's First Pixel Address     S -- set Window's Monitor Position
X -- set H/W Pixel Size                    R -- Regenerate Screen
                                           Q -- Quit Customization
                        ---- GSP Memory Definitions ---
 General Memory: Begin     End          Screen Memory: Begin      Pitch (pixels)
                 00000000  0041C000                   00000000            256

                      ---- GSP Screen Display Area ----
          First Displayed Pixel          Pixel Size     Display Dimensions
                -X-          -Y-              4            Width   Height
                  0            0                            256     128

                      ---- Host Window Area ----
       GSP POsition       Host Dimensions        Monitor Screen Position
         -X-    -Y-        Width   Height              -X-     -Y-
           0      0          720      276               386     170
 Choice:
```

Figure 11-22. Graphics Environment Menu

Selecting either the S or G menu allows you enter changes for each of these values. Entering a carriage return instead of a value leaves the value unchanged.

Syntax **MM***[+] address { 16-bit-value | 32-bit-value | assembler-statement }*

Description The MM command allows you to modify or interrogate memory. The *address* is specified by a 32-bit address. The default format of the *address* is hexadecimal. The *address* must be word aligned; if it is not, the simulator word-aligns it by setting the four LSBs to 0.

The second parameter is optional and can be:

- A *16-bit value,*
- A *32-bit value,* **or**
- A line of TMS34010 *assembler code.*

The default format for the *value* parameters is hexadecimal. The default for values specified by assembler code is hexadecimal, except for the TRAP, SETF, and K instructions. If you follow the command with the optional + symbol, the simulator increments the address by the size of the data you enter.

If you do not use a *value* parameter, then the command reports on the contents of the memory location. The *address* is displayed in hexadecimal, decimal, and as an XY address. The contents of the word at the address in memory are also displayed as hexadecimal, decimal, and as a disassembled instruction.

Example 1 Use the command to report on a memory location. Assume the following initial conditions:

Memory location 0FF0 contains 2980h
CONVSP = 15h
CONVDP = 16h
OFFSET = 0
PSIZE = 4

```
Command[1] MM FF8 CR
```

This example produces the display shown in Figure 11-23.

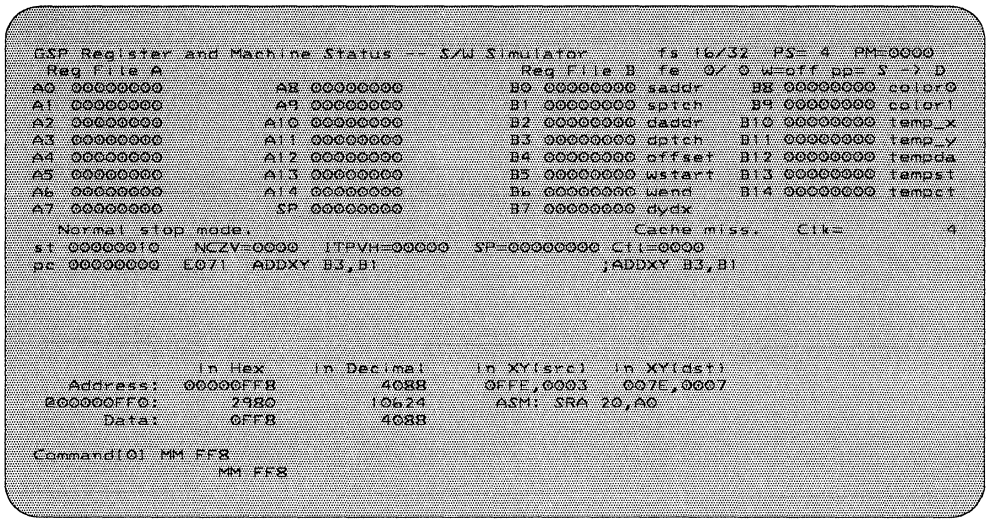


Figure 11-23. Modify Memory Display

This form of the command overwrites any trailing commands on the command line; trailing commands are not executed. You can use this command to find equivalent linear addresses from XY addresses, although the V command is also provided for this purpose.

Example 2 Use the MM command to modify a memory location:

```
Command[1] MM FF8 FEC4 CR          (16-bit value)
Command[1] MM FF8 1FEC4 CR         (32-bit value)
Command[1] MM FF8 MOVE A0,B9 CR    (Assembler code)
```

Each of these examples changes the value of the word or words starting at address 0FF8h to the type of value on the right. Note that a 16-bit value or 32-bit value is specified indirectly by the number of hexadecimal digits required to hold the result:

- FEC4 is a 16-bit value
- 1FEC4 is a 32-bit value.

You can force a hexadecimal value to be a 32-bit value by using leading 0s. The value 0FEC4 is a 32-bit value. Values specified with a decimal format override take up as much space as required to hold the hexadecimal equivalent, but leading 0s *are not* taken into account. For negative numbers, the space is calculated for the positive equivalent. Thus, -1 is a 16-bit value.

Specifying a line of assembler code modifies as many words as it takes for the opcode and its operands to be placed in memory. This can be as many

as five words. All values in the assembler code specification must be numeric values as opposed to symbolic. For address relative instructions, the value is specified as the address. The line assembler calculates the relative offset for you. Except for the requirement that values cannot be symbolic, the syntax of assembly code for the line assembler is the same as described in the assembly language section.

Note that if the graphics display is enabled and the memory location is in screen memory and is in the host window, you may see a change in the graphics display. The change in memory is treated in the same way as if it had been done through program execution.

Syntax **MMF***[+] address field-value field-size*

Description The MMF command modifies memory on not necessarily word-aligned boundaries using a specified field size. The *address* is specified by either a 32-bit bit address or an XY address. The default format of *address* is hexadecimal. The second parameter, *field value*, is a field of one to 32 bits. The default format for *field value* is hexadecimal. The value for *field size* is from one to 32. The default format for *field size* is decimal. If you follow the command with the optional + symbol, the simulator increments the address by the size of the data you enter.

Example Modify a memory field:

```
Command[1] MMF FF8 F 4 CR
Command[1] MMF FF8 %100 7 CR
Command[1] MMF FF8 1 1 CR
```

Each of these examples changes the value of the field starting at address 0FF8h to the value on the right. If the specified field value is larger than the field size, then the low order bits up to the field size are inserted. That is, the LSBs of the value are placed into the field in memory.

Note that if the graphics display is enabled and the memory location is in screen memory and is in the host window, you may note a change in the graphics display. The change in memory is treated in the same way as if it had been done through program execution.

Syntax **MT** *[{ OFF | { 27 | 28 | 29 } }]*

Description The MT command modifies the simulator's treatment of the three special traps:

27 converts TMS34010 address for use by the simulator's print utility.

28 uses the simulator's PRINTF utility.

29 terminates execution and return to the simulator's user level.

The optional *OFF* parameter turns off the processing of all the special traps; they are then treated as normal traps. The specification of the optional trap number turns the specified special trap back on. You can only specify one of the three special traps listed above. If you do not supply any parameter, then the simulator displays the status of the three special traps.

Figure 11-24 illustrates the MT display.

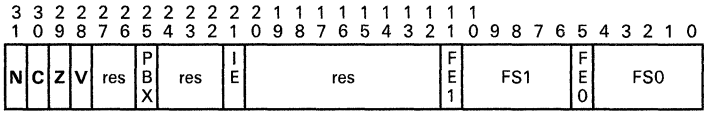
```
GSP Register and Machine Status -- S/W Simulator      fs 16/32  PS= 4  PM=0000
  Reg File A                                  Reg File B  fs  0/ 0 W=off pp= S -> D
A0 00000000          A8 00000000          B0 00000000 saddr     B8 00000000 color0
A1 00000000          A9 00000000          B1 00000000 sptch     B9 00000000 color1
A2 00000000          A10 00000000         B2 00000000 daddr     B10 00000000 temp_x
A3 00000000          A11 00000000         B3 00000000 dptch     B11 00000000 temp_y
A4 00000000          A12 00000000         B4 00000000 offset    B12 00000000 tempda
A5 00000000          A13 00000000         B5 00000000 wstart    B13 00000000 tempst
A6 00000000          A14 00000000         B6 00000000 wwnd      B14 00000000 tempct
A7 00000000          SP 0004DA00          B7 00000000 dydx
  Normal stop mode.                                  Cache off.     Clk=      23653
st 20200010   NCZV=0010  ITPVH=11000  SP=0004DA00 Ctl=8020
pc 000202B0   0D3F  CALLR 21SA0                   ;TRAP 29


                    Convert addresses trap (27) is: on
                    Printf call trap (28) is: on
                    End simulation trap (29) is: on




 Command(0) MT
            MT
```

Figure 11-24. Modify Special Traps Display

Syntax **NCZV** *[4-bit-value]*

Description The NCZV command allows you to display or modify the contents of the NCZV status bits.

N sign bit (status bit 31)
C carry bit (status bit 30)
Z zero bit (status bit 29)
V overflow bit (status bit 28)



Status Register

To modify any of the bits, enter a 4-bit value composed of 0s and 1s. To display the current contents of these bits, enter the command without any value.

Example 1 Set the sign bit to 1 and the other bits to 0s:

```
Command [0] NCZV 1000 CR
```

Example 2 Display the contents of the NCZV bits:

```
Command [0] NCZV CR
Command [0] NCZV = 1000
```

Syntax **NR** *register name*

Description The NR command allows you to assign a name to any of the general-purpose registers or the stack pointer.

- The *register* parameter can be A0–A14, B0–B14, or SP.
- The *name* is a 1- to 6-character alternate name for the register. The register name is used in the machine-state display in reverse assemblies (it appears next to the register in the screen display). It cannot work with the register-value exchange designation.

Example 1 Designate register A0 as SUM:

```
Command [1] NR A0 SUM CR
```

SUM can now be substituted for A0. Also, in reverse assemblies, SUM is used instead of A0 (for example, `MOVE A4,A0` appears as `MOVE A4,SUM`).

Example 2 To delete SUM as the name for A0, enter:

```
Command [1] NR A0 CR
```

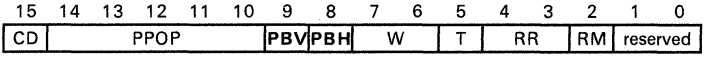
Syntax

PBH [{ 0 | 1 }]

PBV [{ 0 | 1 }]

Description

These commands allow you to set, reset, or toggle the PBH or PBV bit to select the horizontal and vertical directions for PIXBLTs. The PBH and PBV bits are the PIXBLT horizontal and vertical direction bits (bit 8 and bit 9 in the CONTROL register):



CONTROL Register

The optional *0|1* parameter has the following effects:

PBH=0 increment X (move from left to right)

PBH=1 decrement X (move from right to left)

PBV=0 increment Y (move from top to bottom)

PBV=1 decrement Y (move from bottom to top)

If you do not use the 0|1 parameter, then the selected bit is toggled. Note that the value of both of these bits are shown in the the machine-state display as the H and V bits in the ITPVH field.

Example

Set the PBV bit:

```
Command[1] PBV 1 CR
```

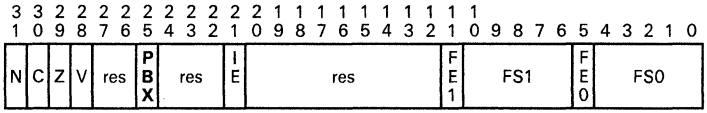
This sets the PBV bit to 1, which causes PIXBLT instructions to decrement in the Y direction. If you then enter:

```
Command[1] PBV CR
```

This toggles the PBV bit, which causes PIXBLT instructions to increment in the Y direction.

Syntax **PBX** *[{0|1}]*

Description The PBX command allows you to set, reset, or toggle the PBX bit. The PBX bit is the PixBlt executing bit (bit 25 in the status register):



Status Register

The optional *0|1* parameter has the following effects:

PBX=0 PIXBLT execution is not resumed after an interrupt.

PBX=1 PIXBLT execution is resumed after an interrupt.

If you do not specify a 0 or a 1, the value of the PBX bit is toggled. Note that this bit is displayed in the ITPVH field of the machine-state display.

Example Set the PBX bit to 1:

```
Command[1] PBX 1 CR
```

This causes an RETI instruction to resume execution of a PixBlt instruction. Of course, if a PixBlt instruction was not in progress, then unpredictable results occur. If you then enter:

```
Command[1] PBX CR
```

This toggles the PBX bit back to 0, so that any interrupted PIXBLT is not resumed.

Syntax **PC** *[32-bit-value]*

Description The PC command allows you to modify or display the contents of the program counter. If you use the optional *32-bit value*, then the simulator replaces the contents of the PC with this value. The default type for the replacement value is hexadecimal.

Note that the PC always contains a word aligned value; that is, the lower 4 bits are 0. If the value is not word aligned, the simulator word-aligns it by truncating its lower 4 bits truncated to 0 before loading it into the PC.

If you do not specify a new value for the PC, the simulator displays the current contents of the PC.

Example 1 Modify the contents of the PC:

```
Command[1] PC 4302 CR
```

The PC now contains the value 00004300h. Note the truncation of the lower 4 bits of the value.

Example 2 Display the contents of the PC from the command line by using the command without specifying a value:

```
Command[1] PC CR
Command[1] PC = 00004300
```

The contents of the PC are now visible in the command buffer. This is useful for viewing the contents of the PC while the text display is off. Note that this form of the command destroys any monitor commands that follow in the same buffer.

Syntax **PM** *[16-bit-value]*

Description The PM command allows you to modify or display the contents of the PMASK register. If you specify a *16-bit value*, the simulator replaces the conentes of the PMASK register with this value. The default type for the value is hexadecimal.

If you do not specify a new value for the PMASK register, the simulator displays the current contents of the PMASK register.

Example 1 Modify the contents of the PMASK register:

```
Command[1] PM FFFE CR
```

The PMASK register now contains the value 0FFFEh. This value allows only the LSB of each word written during graphics instructions to be affected.

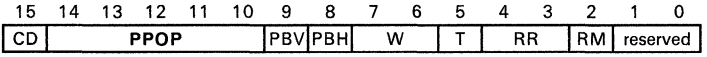
Example 2 Display the contents of the PMASK register from the command line:

```
Command[1] PM CR
Command[1] PM = FFFE
```

The contents of the PMASK register are now visible in the command buffer. This is useful for viewing the contents of the PMASK register while the text display is off. Note that this form of the command destroys any monitor commands that follow in the same buffer.

Syntax **PP** *[pixel-processing-option]*

Description The PP command allows you to modify or display the contents of the PPOP bits. The PPOP bits are the pixel-processing option bits (bits 10–14 in the CONTROL register):



CONTROL Register

If you specify a *pixel processing option,* the simulator stores this value in the PP bits. The default type for the pixel processing option is decimal.

If you do not specify a new pixel processing option, the simulator displays the current pixel processing option.

Table 11-6 lists valid pixel processing options.

Table 11-6. Pixel Processing Options

PP Bits	Operation	Description
0	S → D	Replace destination with source
1	S AND D →D	AND source with destination
2	S AND $\overline{D}$ →D	AND source with NOT destination
3	0 → D	Replace destination with 0s
4	S OR $\overline{D}$ →D	OR source with NOT destination
5	S XNOR D →D	XNOR source with destination
6	$\overline{D}$ →D	Negate destination
7	S NOR D →D	NOR source with destination
8	S OR D →D	OR source with destination
9	D → D	No change in destination
10	S XOR D →D	XOR source with destination
11	$\overline{S}$ AND D → D	AND NOT source with destination
12	1 → D	Replace destination with 1s
13	$\overline{S}$ OR D → D	OR NOT source with destination
14	S NAND D →D	NAND source with destination
15	$\overline{S}$ → D	Replace destination with NOT source
16	D + S → D	Add source to destination
17	D ADDS S → D	Add source to destination with saturation
18	D - S → D	Subtract source from destination
19	D SUBS S → D	Subtract source from destination with saturation
20	D MAX S → D	Maximum of source and destination
21	D MIN S → D	Minimum of source and destination

Example 1 Modify the contents of the PP bits:

```
Command[1] PP 10 CR
```

The PP bits now contain the value 10.

Example 2 Display the contents of the PP bits from the command line:

```
Command[1] PP CR
Command[1] PP = 10
```

The contents of the PP bits are now visible in the command buffer. This is useful for viewing the contents of the PP bits while the text display is off. Note that this form of the command destroys any monitor commands that follow in the same buffer.

Syntax **PS** *[pixel-size]*

Description The PS command allows you to modify or display the contents of the PSIZE register. If you specify a replacement *pixel size*, then the simulator replaces the contents of the PSIZE register with the new pixel size. The default type for the pixel size is decimal.

If you do not specify a new pixel size, the simulator displays the current contents of the PSIZE register.

Example 1 Modify the contents of the PSIZE register:

```
Command[1] PS 8 CR
```

The PSIZE register now contains the value 8. This value causes the simulator to process pixels at a size of 8 bits per pixel. Note that the only valid values for the pixel size are 1, 2, 4, 8, and 16. If you specify any other value, the simulator will issue an error message.

Example 2 Display the contents of the PSIZE register from the command line:

```
Command[1] PS CR
Command[1] PS = 8
```

The contents of the PSIZE register are now visible in the command buffer. This is useful for viewing the contents of the PSIZE register while the text display is off. Note that this form of the command destroys any monitor commands that follow in the same buffer.

Syntax **Q**[*][**C**][**S**]

Description The Q command terminates the simulator session.

When you execute the Q command, the simulator asks if you are sure that you want to terminate the session. If you answer yes, then all files that the simulator has opened are closed and the simulator terminates execution.

You can use three parameters with this command, singly or in combination.

* The simulator does not ask you to verify termination. This is useful if you are running the simulator as part of a batch stream and you do not want any keyboard inputs. (See also the -f option.)

C The simulator clears the screen of both text and graphics on exit.

S The simulator executes the equivalent of the save machine state command with no parameters; that is, the machine state is saved to the file `smsfile.000`.

Example Terminate the simulator session:

```
Command[1] Q  CR
   ARE YOU SURE? [Y/N]:
```

Syntax **RC** *[clock-count]*

Description The RC command begins execution of memory resident TMS34010 code. If you specify a *clock count*, then the simulator executes instructions until the timing clock has progressed *clock count* number of cycles; at this point, the simulator returns control to the monitor. If you do not specify a *clock count*, or if you specify 0 as the *clock count*, then the execution is free run and is equivalent to the RUN command with no run count. The default type for the count is decimal.

You can prematurely terminate the execution by typing a character from the keyboard. Execution also halts if a breakpoint is encountered or if an execution error occurs. Execution will not halt in the middle of an instruction, except for instructions that are interruptible (such as the PixBlt instructions); therefore, the timing clock is usually not incremented by exactly *clock count* number of cycles when execution terminates.

Example Execute code and halt at the nearest interruption point after 1000 (decimal) clock cycles:

```
Command[1] RC 1000   CR
```

Syntax **RDE** *[file-number-extension]*

Description The RDE command restores the saved debugging environment from a file in the default directory. You can use the RDE command in conjunction with the SDE (save debug environment) command to restore the following from a saved debugging environment context:

- Traces
- Breakpoints
- Register names
- Command buffers

If you do not specify a *file number extension,* then the restoration file is called `smsfil.000`.

If you do specify a *file number extension,* then the simulator converts it to the ASCII of its decimal representation and fills it with zeros on the left to form a 3-character file extension. This limits the *file number extension* to 0–999, inclusively. The simulator uses this 3-character extension to form the filename `smsfil.nnn`, where *nnn* is the 3-character file extension. The default type for the file number extension is decimal.

To restore a specific debugging environment via the RDE command, you must specify the same file number extension that it was saved with. See the SDE command for more information.

Example 1 Restore the debugging environment from data stored in the file `sdefil.042`:

```
Command[1] RDE 42 CR
```

Example 2 Restore the debugging environment from data stored in the default file `sdefil.000`:

```
Command[1] RDE CR
```

Syntax **REset**

Description The REset command performs a simulated reset of the TMS34010 via a monitor command. This initializes the program counter with the contents of trap vector 0, sets the status register to 10h, sets all IO registers to 0, removes all pending interrupts, and clears the cache.

Example

```
Command[1] RE CR
```

or

```
Command[1] RESET CR
```

Syntax **RGE** *[file-number-extension]*

Description The RGE command restores the simulated graphics environment from a file in the default directory. You can use the RGE command in conjunction with the SGE (save graphics environment) command to restore a previously saved graphics environment context.

If you do not specify a file number extension, then the restoration file is called `smsfil.000`.

If you do specify a *file number extension*, then the simulator converts it to the ASCII of its decimal representation and zero fills it on the left to form a 3-character file extension. This limits the *file number extension* to 0–999, inclusively. The simulator uses this 3-character extension to form the filename `smsfil.nnn`, where *nnn* is the 3-character file extension. The default type for the file number extension is decimal.

To restore a specific graphics environment via the RGE command, you must specify the same file number extension that it was saved with. The SGE command discusses saving the graphics environment.

The graphics environment saved via the SGE command includes the general memory beginning and ending values, the screen memory beginning and ending values, the TMS34010 upper left screen origin, and other elements described by the G command.

Example 1 Restore the graphics environment from data stored in the file `sgefil.042`:

```
Command[1] RGE 42 CR
```

Example 2 Restore the graphics environment from data stored in the default file `sgefil.000`:

```
Command[1] RGE CR
```

Syntax **RIO**

Description The RIO command restores the contents of the I/O registers from a copy that is kept in simulator local memory (as opposed to on disk). The copy of the I/O registers should have been save previously with the SIO (save I/O registers) command. If the registers were not saved previously, RIO sets them to 0.

Example Restore the local copy of the I/O register:

```
Command[1] RIO CR
```

Syntax **RMI** *[file-number-extension [offset]]*

Description RMI restores the region of TMS34010 memory that was previously saved with the SMI command. You can use the RMI command in conjunction with the SMI (save memory image) command to restore a range of memory that were previously saved in a disk file. Note that the beginning and ending addresses of the memory image have been stored in the file along with the memory data, and need not be specified.

If you do not specify a *file number extension*, then the restoration file is called `smsfil.000`.

If you do specify a *file number extension*, then the simulator converts it to the ASCII of its decimal representation and zero fills it on the left to form a 3-character file extension. This limits the *file number extension* to 0–999, inclusively. The simulator uses this 3-character extension to form the filename `smsfil.nnn`, where *nnn* is the 3-character file extension. The default type for the *file number extension* is decimal.

You can offset the memory image from its present location in memory by specifying an *offset* parameter. The default type for the *offset* is hexadecimal. Note that the *offset* is treated as a signed 32-bit value. If you specify an *offset*, then you must also specify a *file number extension*.

To restore a specific memory image via the RMI command, you must specify the same file number extension that it was saved under. The simulator tells you if the file does not exist. If the simulator reaches a premature end-of-file condition on the *smifil* or if it encounters a memory write error, then the data restored thus far from the file to memory remains in memory. In both cases, the simulator will inform you of the incomplete memory restoration. See the SMI command for saving the memory image.

Example 1 Restore the memory image data stored in the file `smifil.100`:

```
Command[1] RMI 100 CR
```

Example 2 Restore the memory image data stored in the file `smifil.047`, offsetting the data in memory by a value of 0401h bits:

```
Command[1] RMI 47 0401 CR
```

Example 3 Restore the memory image data stored in the default file `smifil.000`:

```
Command[1] RMI CR
```

You could also use the default with an offset, as shown below (780h is the offset):

```
Command[1] RMI 0 780 CR
```

Syntax **RMS** *[file-number-extension]*

Description You can use the RMS command in conjunction with the SMS (save machine state) command to restore a machine state of the simulator from a disk file in the default directory. The machine-state elements restored include:

- General-purpose registers
- I/O registers
- Status register
- PC
- Clock
- Trap vectors

If you do not specify the *file number extension*, then the restoration file is called `smsfil.000`.

If you do specify a *file number extension*, then the simulator converts it to the ASCII of its decimal representation and zero fills it on the left to form a 3-character file extension. This limits the *file number extension* to 0–999, inclusively. The simulator uses this 3-character extension to form the filename `smsfil.nnn`, where *nnn* is the 3-character file extension. The default type for the *file number extension* is decimal.

To restore a specific graphics environment via the RMS command, you must specify the same file number extension as with which it was saved. The simulator informs you if the restoration file does not exist. The SMS command discusses saving the machine state.

Example 1 Restore the machine-state data in the file `smsfil.100`:

```
Command[1] RMS 100 CR
```

Example 2 Restore the machine-state data stored in the default file `smsfil.000`:

```
Command[1] RMS CR
```

Syntax **RR**

Description The RR command restores the contents of the A- and B-file registers from a copy that is kept in simulator local memory (as opposed to on disk). The copy of the registers should have been saved previously with the SR (save registers) command. If the registers were not previously saved then the simulator sets them to 0.

Example Restore the local copy of the A- and B-file registers:

```
Command[1] RR CR
```

Syntax **RS**

Description The RS command regenerates the picture information from the screen memory. This is useful when you have changed the value of any of the parameters that affect the way in which memory is interpreted to the display screen. These parameters include:

- Hardware pixel size,
- Screen memory start and end,
- The displayable window into screen memory,
- Host window start and end, host window location on the display screen, **and**
- Host window dimensions.

You can change these parameters with the G command.

Since the screen regeneration may take a substantial amount of time, depending upon the number of display screen pixels that are affected, the simulator asks you if you are sure that you want to continue. This command can also be invoked in the G command.

Example Regenerate the screen from data supplied via the G command, or indirectly via the RGE command.

```
Command[1] RS CR
```

Syntax **RUn** *[instruction-count]*

Description The RUn commands executes instructions either continuously or until a specified *instruction count* is reached. The screen display is not updated until execution is halted. You can enter the command as RUN or abbreviate it as RU.

If you specify an *instruction count*, the simulator executes *instruction count* number of instructions and then return control to the command line. The default type for the *instruction count* is decimal. If you do not specify an *instruction count*, the simulator executes instructions until:

- You halt execution with a keystroke,
- An error is encountered,
- A breakpoint is encountered, **or**
- A TRAP 29 is executed.

You can use the MT command to disable the halt on TRAP 29.

Example 1 Execute the RUN command with an instruction count of 100:

```
Command[1] RUN 100 CR
```

or

```
Command[1] RU 100 CR
```

Execution halts after 100 instructions if none of the halt conditions mentioned above have occurred.

Example 2 Execute the RUN command without an instruction count:

```
Command[1] RUN CR
```

or

```
Command[1] RU CR
```

Execution halts only if one of the halt conditions mentioned above has occurred.

Also see the RC, SS, BP, and MT commands.

Syntax **SDE** *[file-number-extension]*

Description The SDE command saves the current simulator debugging environment to a file in the default directory. Debugging elements that are saved include:

- Traces
- Breakpoints
- Register names
- Command buffers

You can use the RDE command to restore the saved debug environment.

If you do not specify a file number extension, then the file is called `sdefil.000`.

If you do specify a *file number extension*, the simulator converts it to the ASCII of its decimal representation and zero fill it on the left to form a 3-character file extension. This limits the *file number extension* to 0–999, inclusively. The simulator uses this 3-character extension to form the file name `sdefil.nnn`, where *nnn* is the 3-character file extension. The default type for the *file number extension* is decimal.

To restore a specific debug environment via the RDE command, you must specify the same file number extension as with which it was saved.

Example 1 Save the debugging environment in the file `sdefil.043`:

```
Command[1] SDE 43 CR
```

Example 2 Save the debugging environment in the default file `sdefil.000`:

```
Command[1] SDE CR
```

Syntax **SF** *filename*

Description The SF command displays the contents of a file (specified by *filename*) to the screen. This allows you to access system files without corrupting or losing the current simulation. The simulator clears the screen before and after displaying the file.

If the specified file is longer than 23 lines, the simulator displays 23 lines and pause until the you enter a carriage return. If you want to halt the display, enter a **Q**.

This command is useful for displaying assembly listing and linker map files during a debugging session.

Example Display the contents of the file `example.lst`:

```
Command[1] SF EXAMPLE.LST CR
```

Syntax **SGE** *[file-number-extension]*

Description The SGE command saves the simulated graphics environment to a file in the default directory. The graphics elements saved by the SGE command include:

- General memory starting and ending values
- Screen memory starting and ending values
- Upper left screen origin

You can use the RGE command to restore a saved graphics environment.

If you *do not* specify a *file number extension*, then the file is called `sgefil.000`. If you *do* specify a *file number extension*, the simulator converts it to ASCII of its decimal representation and zero fills it from the left to form a 3-character file extension. This limits the *file number extension* to 0–999, inclusively. The simulator uses this 3-character extension to form the file name `sgefil.nnn`, where *nnn* is the 3-character file extension. The default type for the *file number extension* is decimal.

To restore a specific graphics environment via the RGE command, you must specify the same file number extension that is was saved under.

Example 1 Save the graphics environment in the file `sgefil.043`:

```
Command[1] SGE 43 CR
```

Example 2 Save the graphics environment in the default file `sgefile.000`:

```
Command[1] SGE CR
```

Syntax **SIO**

Description The SIO command saves the contents of all of the I/O registers to a copy that is kept in simulator local memory (as opposed to on disk). Note that this is temporary memory and is cleared between invocations of the simulator. You can restore the I/O registers by using the RIO (restore I/O registers) command.

Example Save a local copy of the I/O registers:

```
Command[1] SIO CR
```

Syntax **SMI** *start-address end-address [file-number-extension]*

Description You can use the SMI command to save a range of memory to disk. The SMI command saves the region of TMS34010 memory from the *start address* to the *end address* in binary format, in a file on disk in the default directory. The default format for both addresses is hexadecimal.

The RMI command restores the memory image.

> **Note:**
>
> The addresses specified for the SMI command are inclusive bit addresses. Thus, if you specify the starting and ending addresses as the same address, then a single bit is saved. If you want to save all of the words of memory from 0 up to and including the word starting at 400h, then the starting and ending addresses should be 0 and 40Fh. If you specify 400h as the ending address, then only the first bit of the word at 400h is saved.

If you do not specify a file number extension, then the save file is called `smifil.000`.

If you do specify a file number extension, the simulator converts it to the ASCII of its decimal representation and zero fill it from the left to form a 3-character file extension. This limits the file number extension to 0–999, inclusively. The simulator uses this 3-character extension to form the file name `smifil.nnn`, where *nnn* is the 3-character file extension. The default type for the file number extension is decimal.

If the save file cannot be created or there is an error while writing to the file (running out of disk space), the saving of memory to the file terminates and the file is closed. If you attempt to restore the memory image in the file, then whatever was stored in the file up to the error is restored. The RMI command then detects a premature end of file on the restoration file and signals an error.

To restore a specific graphics environment via the RMI command, you must specify the same file number extension that it was saved under. See the RGE command for restoring the graphics environment.

You can use the SMI command to preserve a specified memory context for debugging or to store screen data.

Example 1 Save the memory image data from address 1A0h to 200Fh in the file `smifil.792`. The data is stored in noncompressed, binary-image format.

```
Command[1] SMI 1A0 200F 792 CR
```

Example 2 Save a single bit of memory at 1A1h in the file `smifil.003`. The %3 indicates a decimal 3. Since the default type for the extension is decimal, the %3 is equivalent to 3.

```
Command[1] SMI 1A1 1A1 %3 CR
```

Example 3 Save the memory image data from address 440Ch to 4601h in default file `smifil.000`:

```
Command[1] SMI 440C 4601 CR
```

Syntax **SMS** *[file-number-extension]*

Description The SMS command saves the machine state to a file in the default directory. Machine-state elements that are saved include:

- General-purpose registers
- I/O registers
- Status register
- Program counter
- Clock
- Trap vectors

You can use the RMS command to restore the saved machine state.

If you do not specify a file number extension, then the save file is called `smsfil.000`.

If you specify a *file number extension*, the simulator converts it to the ASCII of its decimal representation and zero fills it on the left to form a 3-character file extension. This limits the *file number extension* to 0–999, inclusively. The simulator uses this 3-character extension to form the file name `smsfil.nnn`, where *nnn* is the 3-character file extension. The default type for the *file number extension* is decimal.

To restore a specific graphics environment via the RMS command, you must specify the same file number extension that it was saved under. The RMS command discusses restoring the machine state.

Example 1 Save the status data in the file `smsfil.100`:

```
Command[1] SMS 100 CR
```

Example 2 Save the machine-state data in the default file `smsfil.000`:

```
Command[1] SMS CR
```

Syntax **SP** *[32-bit-value]*

Description The SP command modifies or displays the contents of the stack pointer register. If you specify the optional *32-bit value*, this value replaces the contents of the stack pointer. The default type for this value is hexadecimal. (To modify or display the A- and B-file register, see the A*n* and B*n* commands.)

Example 1 Modify the contents of the stack pointer:

```
Command[1] SP 4000 CR
```

The SP register now contains the value 00004000h.

Example 2 Display the stack pointer contents from the command line:

```
Command[1]SP CR
Command[1] SP = 00004000
```

The contents of the SP register are now visible in the command buffer. This is useful for viewing the contents of the SP register while the text display is off. Note that this form of the command destroys any monitor commands that follow in the same buffer.

Syntax **SR**

Description The SR command saves the contents of the A- and B-file registers in a copy that is kept in the simulator local memory (as opposed to on disk). You can restore the registers from the copy by using the RR (restore registers) command. Note that you can only save one copy of the registers at a time; reinvoking the SR command overwrites the registers that were previously saved.

Example Save a copy of the A and B registers:

```
Command[1] SR CR
```

Syntax

SS *[instruction-count]*

SSF *[instruction-count]*

SSU *[instruction-count]*

SSFU *[instruction-count]*

Description

The SS command allows you to single step through a program for *instruction count* number of instructions, with or without **F**ast update and/or **U**nassembly.

If none of the optional parameters, including F and U, are specified, then the simulator executes only one assembly language instruction and then updates the machine-state display. If you specify the optional *instruction count*, then the simulator executes that number of instructions and completely updates the machine-state display after executing each instruction.

The F and U options allow you to select a fast update or an unassembly:

- A **F**ast update is used when stepping for a number of instructions. The F option inhibits the update of the simulator machine-state display after each instruction except the last. This is functionally equivalent to the RUN command with an instruction count, but executes slightly slower.
- An **U**nassembly provides a 5-line reverse-assembly after each instruction. The reverse-assembly includes information about the two previous program counter locations, the current program counter location, and the two following locations. The display is similar to that of the U command.

You can use the F and U options together to provide a faster single step with unassembly.

Example 1

Single step for one instruction:

```
Command[1] SS CR
```

Example 2

Single step for 10 instructions:

```
Command[1] SS 10 CR
```

Example 3

Use the F and U options:

```
Command[1] SSF CR
Command[1] SSFU 100 CR
Command[1] SSU 10 CR
```

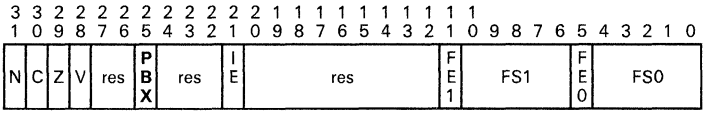
Note that you can use the F, U, and instruction count options independently or in conjunction with one another.

Also see the RC, RUN, MT, and BP commands.

Syntax **ST** *[{ { N | C | V | Z } { 0 | 1 } | 32-bit-value}]*

Description The ST command allows you to modify or display the contents of the status register by specifying either:

- A status bit name and a value for that bit **or**
- A 32-bit value to replace the entire contents of the ST. (The default type for this value is hexadecimal.)



Status Register

You can also selectively set or reset the contents of the N (sign), C (carry), Z (zero), and V (overflow) status bits by specifying the bit's letter name and the desired value. Note that the values of these bits are shown in the NCZV field of the machine-state display.

You can also use ST to display the contents of the status register from the command line by using the command without specifying a value. This is useful for viewing the contents of the status register when the text display is off.

Example 1 Modify the contents of the status register using a 32-bit value:

```
Command[1] ST F0000046 CR
```

Example 2 Turn on the Z bit of the status register:

```
Command[1] ST Z 1 CR
```

Note that the space after ST is optional when specifying a particular status bit; it is allowed for clarity.

To turn off the Z bit, enter:

```
Command[1] STZ 0 CR
```

Example 3 Display the contents of the status register:

```
Command[1]ST CR
Command[1] ST = F0000046
```

The contents of the status register are now visible in the command buffer. Note that this form of the command destroys any monitor commands that follow in the same buffer.

Syntax **SWitch**

Description The SWITCH command modifies the command entry source from the keyboard to the file `gspinput.000`. Commands are accepted as they occur in the file until a SWITCH command is encountered in the file or an EOF is encountered. At this point, control returns to the keyboard.

> **Note:**
>
> If the SWITCH command is interrupted before completing the command string (for example, an unexecutable command is encountered) and terminates with an error message, the string can be continued *at the command after the one in error* by issuing another SWITCH command. If you instead want to begin with the first command in the file, first enter the CIF command.

When the SWITCH command is encountered in the file, the simulator returns to accepting its input from the keyboard. If you enter another SWITCH command, then the simulator accepts input from the file, continuing where it had previously left off in reading the file. If an EOF is encountered then the input file is closed. Another SWITCH command will then begin reading again from the top of the file. You can also cause the simulator to automatically begin reading from the input file by specifying the S option when the simulator is invoked.

Example Switch the command input source:

```
Command[1] SWITCH  CR
```

Figure 11-25 shows a sample input file.

```
re
ssu  13
bpaa  13
rc  1200
switch
```

Figure 11-25. gspinput.000 Example File

Note that the file contents are automatically converted to uppercase.

Syntax **SY** *string*

Description The SY command executes MS-DOS system functions from the simulator. You can use SY to edit, assemble, link, copy files, and perform other functions without exiting the simulator session.

> **Caution:**
>
> **The simulator does not protect your local memory when executing system commands. You should save any nonrecoverable memory contents with the SMI command before executing a system command.**

Example

```
Command[1]  SY COPY \GSP_ASM\HELLO.OBJ HELLO2.OBJ CR
```

or

```
Command[1]  SY gspasm \gsp_asm\hello.obj CR
```

or

```
Command[1]  SY edit \gsp_asm\hello.asm CR
```

or

```
Command[1] SY cd \gsp_asm CR
```

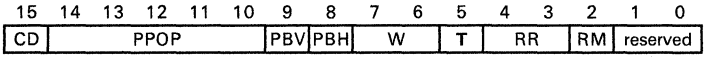
or

```
Command[1] SY dir B: CR
```

The simulator machine-state display is cleared and the MS-DOS command is executed. After the command is complete, the simulator waits for a carriage return before clearing the screen and rebuilding the simulator machine-state display. Normal system control characters affect the execution in the same manner as if the command were invoked from the operating system.

Syntax **T** *[{ 0 | 1 }]*

Description The T command sets, resets, or toggles the contents of the T bit. The T bit is the transparency bit (bit 5 in the CONTROL register):



CONTROL Register

The *0|1* parameter is optional; if you do not specify 0 or 1, the command toggles the current value of the T bit. The *0|1* parameter has the following effects:

T=0 disables transparency.

T=1 enables transparency.

Note that the value of this bit is shown in the ITPVH field of the machine-state display.

Example Set the T bit:

```
Command[1] T 1 CR
```

The T bit is set to 1, enabling transparency.

If you then enter:

```
Command[1] T CR
```

the T bit is toggled back to 0, disabling transparency.

Syntax **TR**

Description The TR command displays all existing traces, along with their active/inactive state.

Each trace is assigned a reference number. You can define a combined maximum of 20 traces and breakpoints at one time. The reference numbers specified here are those that are used in conjunction with the TR*n* command to manipulate the state of each trace in the list. Traces are defined and modified using the commands described in the following sections.

Figure 11-26 shows an example of this display.

```
GSP Register and Machine Status -- S/W Simulator      fs 16/32  PS= 4  PM=0000
  Reg File A                          Reg File B  fe  0/ 0 W=off pp= S -> D
A0 00000000          A8 00000000     B0 00000000 saddr     B8 00000000 color0
A1 00000000          A9 00000000     B1 00000000 sptch     B9 00000000 color1
A2 00000000          A10 00000000    B2 00000000 daddr     B10 00000000 temp_x
A3 00000000          A11 00000000    B3 00000000 dptch     B11 00000000 temp_y
A4 00000000          A12 00000000    B4 00000000 offset    B12 00000000 tempda
A5 00000000          A13 00000000    B5 00000000 wstart    B13 00000000 tempst
A6 00000000          A14 00000000    B6 00000000 wend      B14 00000000 tempct
A7 00000000          SP 0004DA00     B7 00000000 dydx
  Normal stop mode.                                Cache off.    Clk=       1975
st 20000010   NCZV=0010  ITPVH=00000  SP=0004DA00 Ctl=8000
pc 00020280  0D3F  CALLR 207E0                    ;TRAP 29
  0 range:00004000-0004120 on all
  1 value:120F on all
  2 value:4400ptrn:B0CE on reads
  3 adr:120FF310 on IAQs
  4 adr:00004400 ptrn:000B0CE on all




Command[0] TR
           TR
```

Figure 11-26. Display of Existing Traces

Syntax **TR*n*** *[{ Clear | OFf | ON | Toggle | Quit }]*

Description The TR*n* command modifies the status of individual traces. The *n* parameter specifies a specific trace. This number is a decimal integer between 0 and 19 or the letter *X* If you specify *X* as the trace number, then all existing traces are affected. You can define a combination of up to 20 traces and breakpoints. The trace reference number is displayed when the trace is defined, and can be viewed with the TR command. Once you have assigned certain trace conditions to a trace number, the number is associated only with those conditions until you clear the specific trace.

You can use the following options with the traces:

Clear Destroys the trace.

OFf Deactivates the trace temporarily but does not destroy it.

ON Reactivates deactivated traces.

Toggle Toggles the current state of a trace. If the trace is deactivated, T reactivates it; if the trace is activated, T deactivates it.

Quit Terminates the command without making any changes.

Only the significant letters of an option (indicated by upper case letters in the preceding list) are processed. This allows you to specify a shorthand version of the option. For example, Clear and C are treated the same. If you do not enter the option on the command line, then a menu is displayed and you are allowed to select the desired option from the menu.

Example 1 Toggle trace 3:

```
Command[1] TR3 TOG  CR
```

or

```
Command[1] TR3 CR
```

The second entry causes the simulator to display the trace and the menu shown in Figure 11-8.

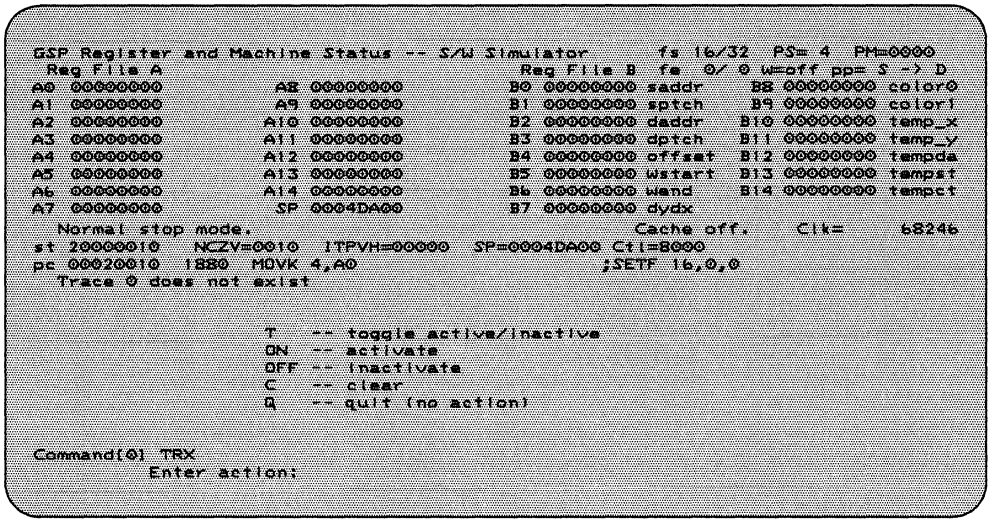


Figure 11-27. Trace Options Display

At this point, you can enter T to toggle the state of trace number 3 to off. Note that trace 3 remains in memory; you can reactivate it with the same command sequence or specify the ON option. Alternatively, you can delete it with the Clear option, and then overwrite it with the TRA, TRD, or TRR commands. The simulator verifies the changes you make to a trace by displaying the trace and its associated conditions in the scratch area.

Example 2 Clear all traces:

```
Command[1] TRX CLEAR CR
```

Syntax

TRAR { *address* | *address-pattern* }

TRAW { *address* | *address-pattern* }

TRAI { *address* | *address-pattern* }

TRAA { *address* | *address-pattern* }

Description

The TRA command sets traces to stop execution when the simulator accesses:

- A specified *address* **or**
- An address that matches a specified *address pattern*.

There are four versions of this command:

TRAR breaks on all *memory reads* from the specified address.
TRAW breaks on all *memory writes* from the specified address.
TRAI breaks on *instruction acquisition* from the specified address.
TRAA breaks on *all memory accesses* to the specified address.

The default type for the address is hexadecimal.

You can use a pattern instead of an address; specify a pattern as a 32-bit binary number with 1s and 0s in the data compare positions and Xs in the don't care positions. You must enclose the pattern in parentheses.

Example 1

Figure 11-28 shows a trace when the simulator fetches from location 120FF310h:

```
GSP Register and Machine Status -- S/W Simulator      fs 16/32  PS= 4  PM=0000
  Reg File A                                Reg File B  fe  0/ 0 w=off pp= S -> D
A0 00000000          A8 00000000          B0 00000000 saddr     B8 00000000 color0
A1 00000000          A9 00000000          B1 00000000 sptch     B9 00000000 color1
A2 00000000         A10 00000000          B2 00000000 daddr    B10 00000000 temp_x
A3 00000000         A11 00000000          B3 00000000 dptch    B11 00000000 temp_y
A4 00000000         A12 00000000          B4 00000000 offset   B12 00000000 tempda
A5 00000000         A13 00000000          B5 00000000 wstart   B13 00000000 tempst
A6 00000000         A14 00000000          B6 00000000 wwnd     B14 00000000 tempct
A7 00000000          SP 0004DA00          B7 00000000 dydx
  Normal stop mode.                                 Cache off.    Clk=       1975
st 2000001C   NCZV=0010  ITPVH=00000  SP=0004DA00 Ctl=B000
pc 00020280  0D3F  CALLR 207E0                   ;TRAP 29
 0 range:00004000-00004120 on all
 1 value:120F on all
 2 value:4400ptrn:B0CE on reads
 3 adr:120FF310 on IAQs




Command[0] TRAI 120FF310
           TRAI 120FF310
```

Figure 11-28. Trace on Address Display

Example 2 Figure 11-29 shows a trace on any memory access from an address that matches the address pattern X1XX0100XX00XXX0:

```
GSP Register and Machine Status -- S/W Simulator      fs 16/32  PS= 4  PM=0000
  Reg File A                                Reg File B  fs  0/ 0 W=off pp= S -> D
A0 00000000          A8 00000000          B0 00000000 saddr     B8 00000000 color0
A1 00000000          A9 00000000          B1 00000000 sptch     B9 00000000 color1
A2 00000000          A10 00000000         B2 00000000 daddr     B10 00000000 temp_x
A3 00000000          A11 00000000         B3 00000000 dptch     B11 00000000 temp_y
A4 00000000          A12 00000000         B4 00000000 offset    B12 00000000 tempda
A5 00000000          A13 00000000         B5 00000000 wstart    B13 00000000 tempst
A6 00000000          A14 00000000         B6 00000000 wend      B14 00000000 tempct
A7 00000000          SP 0004DA00          B7 00000000 dydx
   Normal stop mode.                                  Cache off.     Clk=        1975
st 20000010   NCZV=0010  ITPVH=00000  SP=0004DA00 Ctl=8000
pc 00020280  0D3F  CALLR 207E0                    ;TRAP 29
  0 range:00004000-00004120 on all
  1 value:120F on all
  2 value:4400ptrn:B0CE on reads
  3 adr:120FF310 on IAQs
  4 adr:00004400 ptrn:0000B0CE on all




Command[0] TRAA (X1XX0100XX00XXX0)
           TRAA (X1XX0100XX00XXX0)
```

Figure 11-29. Trace on Address Pattern Display

Syntax

TRDR { *data* | *pattern* }

TRDW { *data* | *pattern* }

TRDI { *data* | *pattern* }

TRDA { *data* | *pattern* }

Description

The TRD command sets traces to stop execution when the simulator accesses:

- A specified word of *data* **or**
- A specified *data pattern*.

There are four versions of this command:

TRDR	breaks on all *memory reads* of the specified data.
TRDW	breaks on all *memory writes* of the specified data.
TRDI	breaks on *instruction acquisition* of the specified data.
TRDA	breaks on *all memory accesses* of the specified data.

The default type for the data is hexadecimal.

You can use a pattern instead of a word of data; specify the pattern as a 16-bit binary number with 1s and 0s in the data compare positions and Xs in the don't care positions. You must enclose the pattern within parenthesis.

Example 1

Figure 11-30 shows a trace on any memory access to data word 120Fh.

```
GSP Register and Machine Status -- S/W Simulator      fs 16/32  PS= 4  PM=0000
  Reg File A                                Reg File B  fs  0/ 0 W=off pp= S -> D
A0 00000000          A8 00000000          B0 00000000 saddr    B8 00000000 color0
A1 00000000          A9 00000000          B1 00000000 sptch    B9 00000000 color1
A2 00000000          A10 00000000         B2 00000000 daddr    B10 00000000 temp_x
A3 00000000          A11 00000000         B3 00000000 dptch    B11 00000000 temp_y
A4 00000000          A12 00000000         B4 00000000 offset   B12 00000000 tempda
A5 00000000          A13 00000000         B5 00000000 wstart   B13 00000000 tempst
A6 00000000          A14 00000000         B6 00000000 wend     B14 00000000 tempct
A7 00000000           SP 0004DA00         B7 00000000 dydx
  Normal stop mode.                                  Cache off,    Clk=       1975
st 2000010   NCZV=0010  ITPVH=00000  SP=0004DA00 Ctl=8000
pc 00020280  0D3F  CALLR 207E0                ;TRAP 29
 0 range:00004000-00004120 on all
 1 value:120F on all

Command[0] TRDA 120F
           TRDA 120F
```

Figure 11-30. Trace on Data Display

Example 2 Figure 11-31 shows a trace on memory reads when the data pattern matches X1XX0100XX00XXX0:

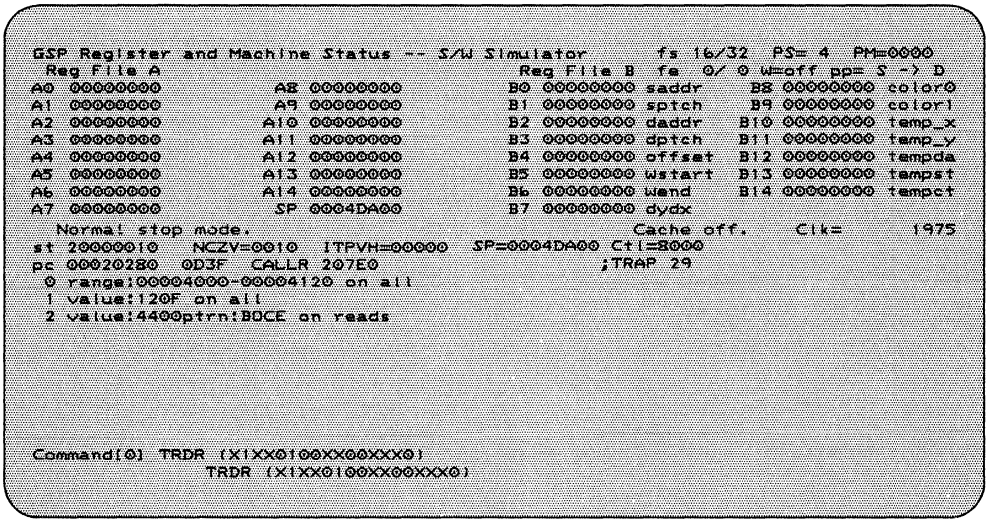


Figure 11-31. Trace on Pattern Display

Syntax

TRRR *start-address end-address*

TRRW *start-address end-address*

TRRI *start-address end-address*

TRRA *start-address end-address*

Description

The TRR command sets traces to stop execution when the simulator accesses memory within a range of addresses. There are four versions of this command:

TRRR	breaks on all *memory reads* in the range.
TRRW	breaks on all *memory writes* in the range.
TRRI	breaks on *instruction acquisitions* in the range.
TRRA	breaks on *all memory accesses* in the range.

The *start address* and *end address* specify the starting and ending points of the address range. The *end address* must be greater than the *start address*, or the simulator issues an error message. Their default type is hexadecimal.

> **Note:**
>
> Note that you cannot specify a pattern for this command.

Example

Figure 11-32 shows a trace on any memory access within in the rage 40000h to 4120h:

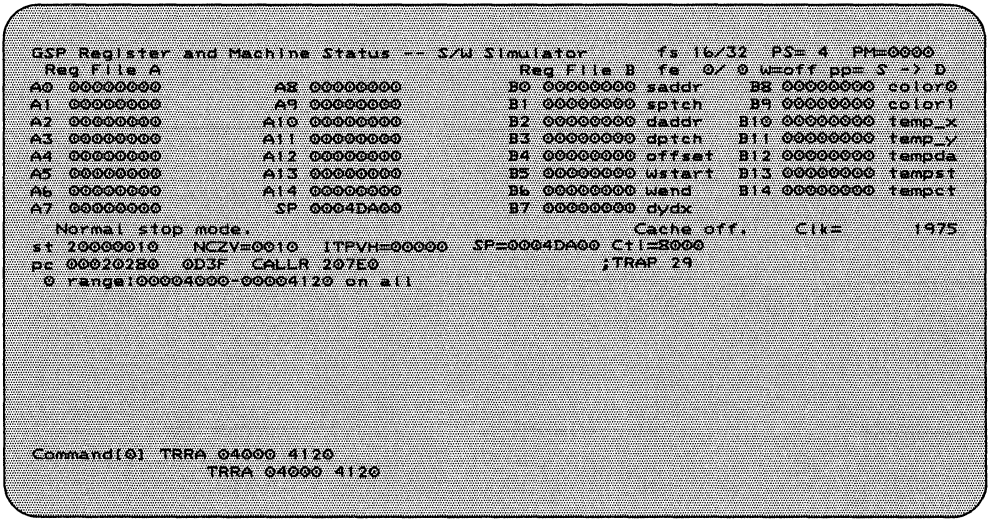


Figure 11-32. Trace on Range Display

Syntax **TX**

Description The TX command toggles the text display (including the simulator machine-state display) so that the graphics display can be clearly seen. Only the command line and the line below it are displayed.

On systems with independent graphics display, such as the TI-PC, the TX command does not affect the graphics display. On systems with integrated text and graphics, such as the IBM-PC, both of the TX and GR commands toggle between text and graphics.

In the same way that the TX command toggles text on and off, the GR command toggles the graphics display on and off. When the text is display off, the command line remains at the bottom of the screen and the simulator status line replaces the previous command entry on the following line.

Note that when you toggle back to text mode, the scratch-display area is cleared.

Example Toggle the text display.

```
Command[1] TX CR

Command[1] TX
  Normal Stop Mode.
```

The monitor status messages now appear below the command line:

```
Command[1] RUN
    Simulator Running ...
```

You can toggle the text back on by repeating the TX command.

Syntax **U** *[start-address] [end-address]*

Description The U command unassembles (reverse-assembles) blocks of memory, depending on whether:

- No addresses are specified,
- Only a *start address* is specified, **or**
- Both a *start address* and an *end address* are specified.

The simulator can display up to nine instructions at one time. If the specified block contains more than nine instructions, enter a carriage return to display the next block of nine instructions.

Example 1 Unassemble without specifying an address:

```
Command[1] U CR
```

Figure 11-33 illustrates this example. The display shows reverse assemblies of:

- The last two program counter locations,
- The current program counter location, **and**
- The next two instructions following the current PC.

```
GSP Register and Machine Status -- S/W Simulator      fs 16/32  PS= 4  PM=0000
  Reg File A                                  Reg File B  fe  0/ 0 w=off pp= S -> D
A0 00000000          A8 00000000          B0 00000000 saddr     B8 00000000 color0
A1 00000000          A9 00000000          B1 00000000 sptch     B9 00000000 color1
A2 00000000         A10 00000000          B2 00000000 daddr    B10 00000000 temp_x
A3 00000000         A11 00000000          B3 00000000 dptch    B11 00000000 temp_y
A4 00000000         A12 00000000          B4 00000000 offset   B12 00000000 tempda
A5 00000000         A13 00000000          B5 00000000 wstart   B13 00000000 tempst
A6 00000000         A14 00000000          B6 00000000 wend     B14 00000000 tempct
A7 00000000          SP 0004DA00          B7 00000000 dydx
  Normal stop mode.                                  Cache off.    Clk=     68246
st 20000010   NCZV=0010  ITPVH=00000   SP=0004DA00 Ctl=8000
pc 00020000  1880  MOVK 4,A0                      ;SETF 16,0,0
     Lnr Addr  Opcode  Revassembly
   pc 000202A0   019D   TRAP 29  st 20000010
   pc 00020000   0550   SETF 16,0,0  st 20000010
   pc 00020010   1880   MOVK 4,A0  st 20000010
   pc 00020020   0580   MOVE A0,@C0000150,0
   pc 00020050   09C0   MOVI >0000,A0
   pc 00020070   0580   MOVE A0,@C0000160,0
   pc 000200A0   09EF   MOVI >4DA00,SP


Command[0] U
  Hit <CR> to continue or "q" to quit:
```

Figure 11-33. Reverse-Assembly Display

Example 2 Figure 11-34 shows an example that unassembles from a starting location.

```
GSP Register and Machine Status -- S/W Simulator      fs 16/32  PS= 4  PM=0000
 Reg File A                                    Reg File B  fe  0/ 0 W=off pp= S -> D
A0 00000000          A8 00000000          B0 00000000 saddr     B8 00000000 color0
A1 00000000          A9 00000000          B1 00000000 sptch     B9 00000000 color1
A2 00000000          A10 00000000         B2 00000000 daddr     B10 00000000 temp_x
A3 00000000          A11 00000000         B3 00000000 dptch     B11 00000000 temp_y
A4 00000000          A12 00000000         B4 00000000 offset    B12 00000000 tempda
A5 00000000          A13 00000000         B5 00000000 wstart    B13 00000000 tempst
A6 00000000          A14 00000000         B6 00000000 wend      B14 00000000 tempct
A7 00000000           SP 0004DA00         B7 00000000 dydx
  Normal stop mode.                                  Cache off.     Clk=       1975
st 20000010   NCZV=0010  ITPVH=00000  SP=0004DA00 Ctl=8000
pc 00020280  0D3F  CALLR 207E0                      ;TRAP 29
     Lnr Addr  Opcode  Revassembly
  pc 00020280   0D3F   CALLR 207E0
  pc 000202A0   091D   TRAP 29
  pc 000202B0   0D3F   CALLR 215A0
  pc 000202D0   091D   TRAp 29
  pc 000202E0   0D5F   CALLA @27300
  pc 00020310   0D3F   CALLR 21740
  pc 00020330   0D3F   CALLR 21920
  pc 00020350   091D   TRAP 29
  pc 00020360   0D5F   CALLA @27590
Command[0] U 20280
  Hit <CR> to continue or "q" to quit:
```

Figure 11-34. Reverse-Assembly from a Starting Location Display

Example 3 Figure 11-35 shows an example that unassembles within a range of memory.

```
GSP Register and Machine Status -- S/W Simulator      fs 16/32  PS= 4  PM=0000
 Reg File A                                    Reg File B  fe  0/ 0 W=off pp= S -> D
A0 00000000          A8 00000000          B0 00000000 saddr     B8 00000000 color0
A1 00000000          A9 00000000          B1 00000000 sptch     B9 00000000 color1
A2 00000000          A10 00000000         B2 00000000 daddr     B10 00000000 temp_x
A3 00000000          A11 00000000         B3 00000000 dptch     B11 00000000 temp_y
A4 00000000          A12 00000000         B4 00000000 offset    B12 00000000 tempda
A5 00000000          A13 00000000         B5 00000000 wstart    B13 00000000 tempst
A6 00000000          A14 00000000         B6 00000000 wend      B14 00000000 tempct
A7 00000000           SP 0004DA00         B7 00000000 dydx
  Normal stop mode.                                  Cache off.     Clk=       1975
st 20000010   NCZV=0010  ITPVH=00000  SP=0004DA00 Ctl=8000
pc 00020280  0D3F  CALLR 207E0                      ;TRAP 29
     Lnr Addr  Opcode  Revassembly
  pc 00020260   06CC   INVALID OP
  pc 00020270   091D   TRAP 29
  pc 00020280   0D3F   CALLR 207E0




Command[0] U 20260 20280
           U 20260 20280
```

Figure 11-35. Reverse-Assembly within a Range of Addresses Display

Syntax **V** *value*

Description The V command displays a specified *value* in several formats:

- The simulator first displays the *value* as if it were an address.
- Next, the simulator displays the contents of the memory word at the address specified by the *value*. (Note that the lower four bits of the value are ignored, since it is treated as a word address.)
- The third line displays the hexadecimal and decimal representations of the *value*.

The default format for the *value* parameter is hexadecimal. The size of *value* can be up to 32 bits.

Example Figure 11-36 shows an example of the V command. Assume the following initial conditions:

Memory location 0FF0 contains 2980h
CONVSP = 15h
CONVDP = 16h
OFFSET = 0
PSIZE = 4

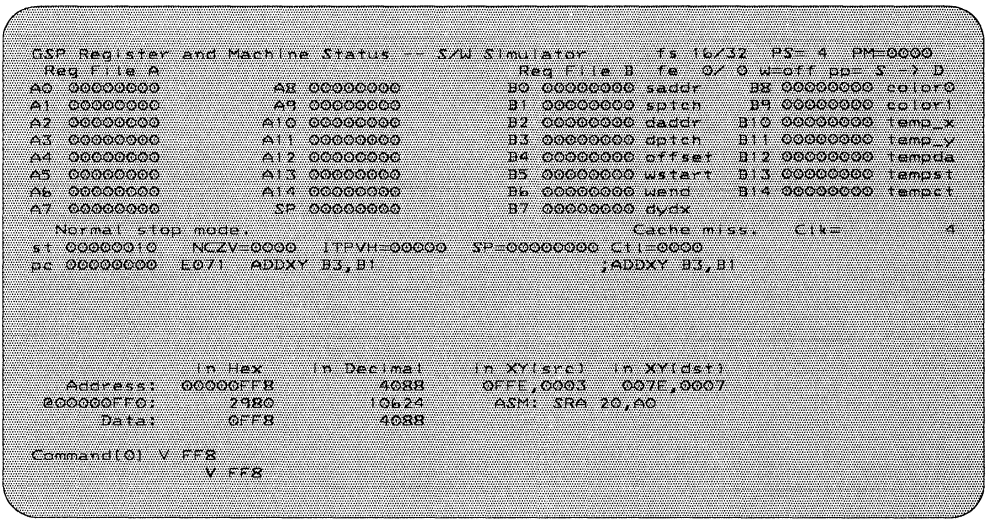
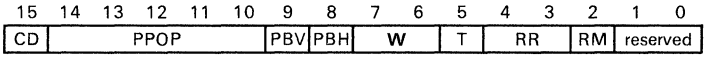


Figure 11-36. Evaluate Data Display

Syntax **W** *[{ 1 | 2 | 3 | 4 }]*

Description The W command sets the window option bits (bits 6 and 7 in the CONTROL register).



CONTROL Register

Setting the W bits to a value of 1, 2, 3, or 4 selects one of the following window options:

W=0 *No windowing.* Writes to any pixel are allowed with no interrupts.

W=1 *Pick function.* Pixel writes are inhibited, and an attempt to write within the window generates an interrupt.

W=2 *Pixel writes within the window are allowed.* An attempt to write outside the window generates an interrupt.

W=3 *Pixel writes outside the window are inhibited,* but no interrupts are generated.

The W field of the machine-state display shows the current windowing option. If you enter the W command and do not select a windowing option, the simulator displays the current windowing option.

Example 1 Modify the contents of the W field:

```
Command[1] W 2 CR
```

Example 2 Display the contents of the W field from the command line:

```
Command[1] W CR
Command[1] W 2
```

The contents of the W field of the CONTROL register are visible in the command buffer.

Syntax **Z**

Description The Z command clears the clock counter (sets it to 0). You can enter the Z command to clear the clock counter at any time during the simulator session. Note that zeroing the clock counter affects interrupts and the host input because they both key on the clock count. Therefore, after clearing the clock count, you should recreate any interrupts or host inputs.

Example Clear the clock counter in the TMS34010 machine-state display:

```
Command[1] Z CR
```

Appendix A

Common Object File Format

The TMS34010 assembler and linker create object files that are in common object file format (COFF). COFF is an implementation of an object file format of the same name that was developed by AT&T for use on UNIX-based systems. This object file format is used because it encourages modular programming and provides more powerful and flexible methods for managing code segments and target system memory.

One of the basic COFF concepts is *sections*. Section 3, Introduction to Common Object File Format, discusses COFF sections in detail. If you understand section operation, you will be able to use the TMS34010 assembly language tools more efficiently.

This appendix contains technical details about COFF object file structure. Much of this information pertains to the symbolic debugging information that is produced by the C compiler. The main purpose of this appendix is to provide supplementary information for those of you who are interested in the internal format of COFF object files.

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 initialized section,
- Relocation information for each initialized section,
- Line number entries for each initialized section,
- 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 does not usually contain relocation entries. Figure A-1 illustrates the overall object file structure.

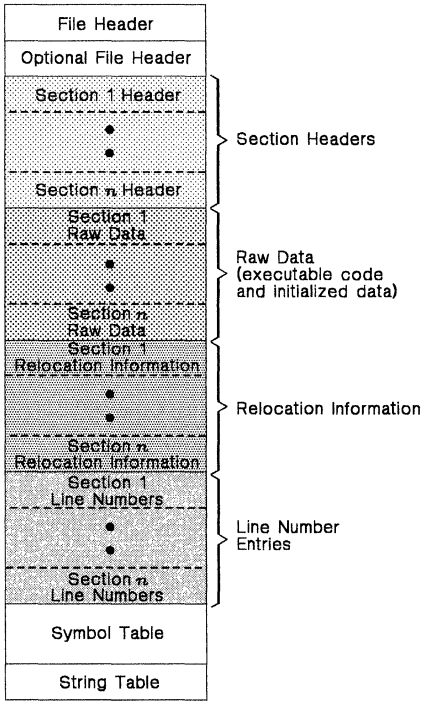


Figure A-1. COFF File Structure

Figure A-2 shows a typical example of a COFF object file that contain the three default sections, .text, .data, and .bss and a named section (referred to as <named>). By default, the .text, .data, and .bss sections, respectively, are placed in the object file, followed by any named sections in the order in which they were encountered. Although uninitialized sections (created with the .bss and .usect directives) have section headers, they have no raw data, relocation information , or line number entries. This is because the .bss and .usect directives simply reserve space for uninitialized data; uninitialized sections contain no actual code.

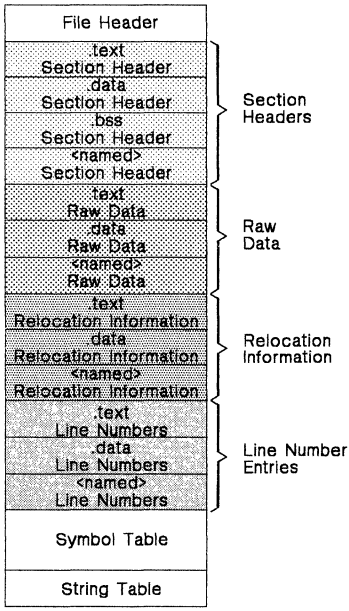


Figure A-2. Sample COFF Object File

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 (090h), indicates that the file can be executed in a TMS34010 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 offset of the symbol table's starting address from the beginning of the file
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 TMS34010 (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 for TMS34010)
2–3	Short integer	Version stamp
4–7	Long integer	Size (in bits) of executable code
8–11	Long integer	Size (in bits) of initialized bits
12–15	Long integer	Size (in bits) of uninitialized data
16–19	Long integer	Entry point
20–23	Long integer	Beginning address of .text
24–27	Long integer	Beginning address of .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 bits (multiple of 16)
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	0002h	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 (relocated, loaded, but not allocated; relocation and line number entries are processed normally)
STYP_TEXT	0020h	Section contains executable code
STYP_DATA	0040h	Section contains initialized data
STYP_BSS	0080h	Section contains uninitialized data
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 the flags field (bytes 36 ane 37) is set to 024h, then STYP_GROUP and STYP_TEXT are both set.

Figure A-3 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.

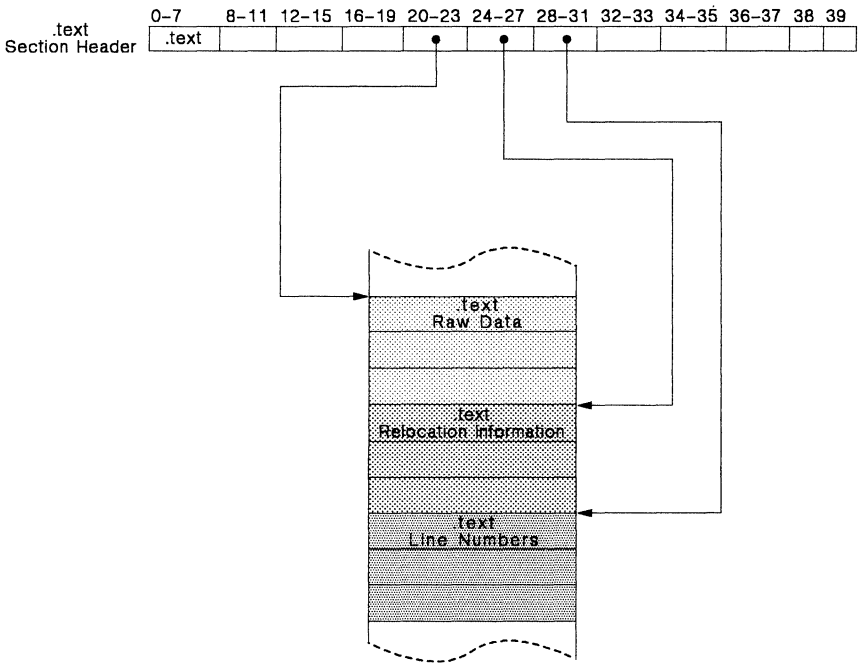


Figure A-3. An Example of Section Header Pointers for the .text Section

As Figure A-2 (page A-3) shows, uninitialized sections (created with the .bss and .usect directives) vary from this format. Although uninitialized sections have section headers, they have no no raw data, no relocation information, no line number information, and occupy no actual space in the object file. Therefore, the number of relocation entries, the number of line number entries, and the file pointers are 0 for an uninitialized section. The section header simply tells the linker how much memory it should reserve for variables.

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 each 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 in the section
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)

- The **virtual address** is the symbol's address in the current section *before* relocation; it specifies *where* a relocation must occur. (This is the address of the field in the object code that must be patched.)

  Here's an example of code that generates a relocation entry:

```
0001                                .global  X
0002 00000000      0D5F             CALLA    X
     00000010 00000000!
```

  In this example, the virtual address of the relocatable field is 10h.

- The **symbol table index** is the index into the symbol table of the relocatable symbol that is referenced. In the preceding example, this field would contain the index of X into the symbol table. The amount of the relocation is the difference between the symbol's current address in the section and its assembly-time address. The relocatable field must be relocated by the same amount as the referenced symbol. In the example, X has a value of 0 before relocation. Suppose X is relocated to address 2000h. This is the relocation amount (2000h - 0 = 2000h), so the relocatable field at address 10h is patched by adding 2000h to it.

  You can determine a symbol's relocated address if you know which section it is defined in. For example, if X is defined in .data, and .data is relocated by 2000h, then x is also relocated by 2000h.

  If the symbol table index ina relocation entry is -1 (0FFFFh), this is called an *internal relocation*. In this case, the relocation amount is simply the amount that the current section is being relocated.

- The **relocation type** specifies the size of the field to be patched and indicates how the patched value should be calculated. The type field depends on the addressing mode that was used to generate the relocatable reference. In the preceding example, the actual address of the ref-

erenced symbol (X) will be placed ina 32-bit field in the object code. This is a 32-bit direct relocation, so the relocation type is R_RELLONG. Table A-7 lists the relocation types.

Table A-7. Relocation Types (Bytes 8 and 9)

Mnemonic	Flag	Relocation Type
R_ABS	0000h	No relocation
R_RELWORD	0010h	16-bit direct
R_RELLONG	0011h	32-bit direct
R_OCRLONG	0018h	1's complement 32-bit direct
R_GSPPCR16	0019h	16-bit relative (in words)

A.6 Line Number Table

The object file contains a table of line number entries that are useful for symbolic debugging. When the C compiler produces several lines of assembly language code, it creates a line number entry that maps these lines back to the original line of C source code that generated them. Each single line number entry contains 6 bytes of information. 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

Figure A-4 shows how line number entries are grouped into 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

Each entry is divided into halves (as shown in Table A-8):

- 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.

The line entry table can contain many of these blocks.

Figure A-5 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-5 shows what the line number entries would look like for this example.

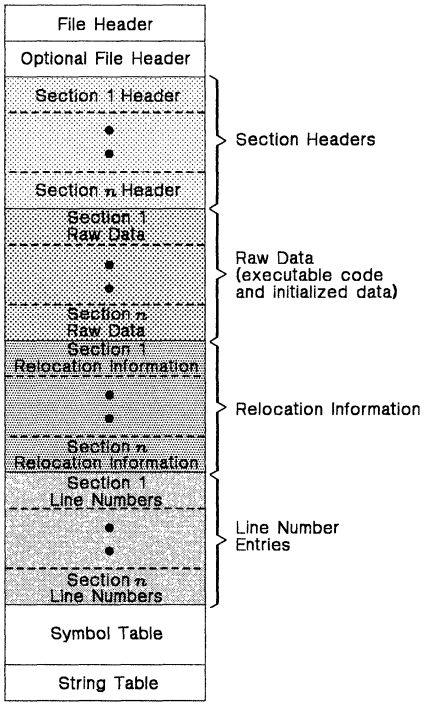


Figure A-5. Line Number Entries Example

(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; this provides 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-6.

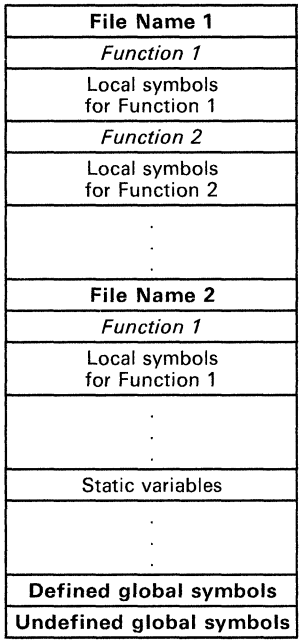


Figure A-6. Symbol Table Contents

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. Each symbol may also have an 18-byte auxiliary entry; the special symbols listed in Table A-10 always have auxiliary entries. Some symbols may not have all the characteristics listed above; if a particular field is not set, it is 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. Each special symbol contains ordinary symbol table information as well as an auxiliary entry. Table A-10 lists these symbols.

Table A-10. Special Symbols in the Symbol Table

Symbol	Description
.file	File name
.text	Address of the .text section
.data	Address of the .data section
.bss	Address of the .bss section
.bb	Address of the beginning of a block
.eb	Address of the end of a block
.bf	Address of the beginning of a function
.ef	Address of the end of a function
.target	Pointer to a structure or union that is returned by a function
.*n*fake	Dummy tag name for a structure, union, or enumeration
.eos	End of a structure, union, or enumeration
_etext, 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.

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.

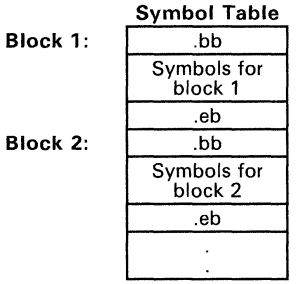


Figure A-7. Symbols for Blocks

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.

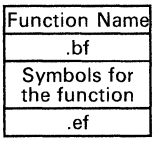


Figure A-8. Symbols for Functions

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-9 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 `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 the 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 (see Table A-18)
.text, .data, .bss	C_STAT	DT_NON	T_NULL	Section (see Table A-19)
tagname	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_NULL	Tag name (see Table A-20)
.eos	C_EOS	DT_NON	T_NULL	End of structure (see Table A-21)
fcname	C_EXT C_STAT	DT_FCN	(See note 1)	Function (see Table A-22)
arrname	(See note 2)	DT_ARY	(See note 1)	Array (see Table A-23)
.bb, .eb	C_BLOCK	DT_NON	T_NULL	Beginning and end of a block (see Table A-24 and Table A-25)
.bf, .ef	C_FCN	DT_NON	T_NULL	Beginning and end of a function (see Table A-24 and Table A-25)
Name related to a structure, union or enumeration	(See note 2)	DT_PTR DT_ARR DT_NON	T_STRUCT T_UNION T_ENUM	Name related to a structure, union, or enumeration (see Table A-26)

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 uninon 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 Filenames

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. Filename Format for Auxiliary Table Entries

Byte Number	Type	Description
0–13	Character	File name
14–17	–	Unused

A.7.8.2 Sections

Table A-19 shows the format of the auxiliary table entries for sections.

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)

Appendix B

Symbolic Debugging Directives

The TMS34010 assembler supports several directives that the TMS34010 C compiler uses for symbolic debugging:

- The **.sym** directive defines a global variable, a local variable, or a function. Several parameters allow you to associate various debugging information with the symbol or function.
- The **.stag**, **.etag**, and **.utag** directives define structures, enumerations, and unions, respectively.
- The **.member** directive specifies a member of a structure, enumeration, or union.
- The **.eos** directive ends a structure, enumeration, or union definition.
- The **.func** and **.endfunc** directives specify the beginning and ending lines of a C function.
- The **.func** and **.endfunc** directives specify the bounds of C blocks.
- The **.file** directive defines a symbol in the symbol table that identifies the current source file name.
- The **.line** directive identifies the line number of a C source statement.

These symbolic debugging directives are not usually listed in the assembly language file that the compiler creates. If you want them to be listed, invoke the code generator with the -o option, as shown below:

```
gspcg -o input file
```

This appendix contains an alphabetical directory of the symbolic debugging directives. Each directive contains an example of C source and the resulting assembly language code.

Syntax **.block** *beginning line number*

.endblock *ending line number*

Description The .block and .endblock directives specify the beginning and end of a C block. The line numbers are optional; they specify the location in the source file where the block is defined.

Note that block definitions can be nested. The assembler will detect improper block nesting.

Example Here is an example of C source that defines a block, and the resulting assembly language code.

C source:

```
      .
      .
      .
{                       /*  Beginning of a block  */
     int  a,b;
     a = b;
}                       /*  End of a block        */
      .
      .
      .
```

Resulting assembly language code:

```
.func   2
.sym    _a,0,4,1,32
.sym    _b,32,4,1,32
.line   3
.endfunc     5
```

Syntax **.file** *"filename"*

Description The .file directive allows a debugger to map locations in memory back to lines in a C source file. *Filename* is the name of the file that contains the orginal C source program. The first 14 characters of the filename are significant; any pathname information is stripped away.

You can also use the .file directive in assembly code to provide a name in the file and improve program readability.

Example Here's an example of the .file directive. The file named *text.c* contained the C source that produced this directive.

```
.file    "text.c"
```

Syntax **.func** *beginning line number*

.endfunc *ending line number*

Description The .func and .endfunc directives specify the beginning and end of a C function. The line numbers are optional; they specify the location in the source file where the function is defined.

Note that function definitions cannot be nested.

Example Here is an example of C source that defines a function, and the resulting assembly language code.

C source:

```
power(x, n)              /*  Beginning of a function */
int x,n;
{
     int i, p;
     p = 1;
     for (i = 1; i <= n; ++i)
          p = p * x;
     return p;           /*  End of function          */
}
```

Resulting assembly language code:

```
0056                                .sym    _power,_power,36,2,0
0057                                .globl  _power
0058
0059                                .func   8
0060
     ****************************************************
0061                        * FUNCTION DEF : _power
0062
     ****************************************************
0063 000002E0              _power:
0064 000002E0      098F             MMTM    SP,A5,A7,A12,FP
     000002F0      050C
0065 00000300      4DCD             MOVE    STK,FP
0066 00000310      0B0E             ADDI    64,STK
     00000320      0040
0067                                .sym    _x,-32,4,9,32
0068                                .sym    _n,-64,4,9,32
0069                                .sym    _i,0,4,1,32
0070                                .sym    _p,32,4,1,32
0071 00000330                       .line   3
0072 00000330      182C             MOVK    1,A12
0073 00000340      B38D             MOVE    A12,*FP(32),1
     00000350      0020
0074 00000360                       .line   4
0075 00000360      838D             MOVE    A12,*FP,1
0076 00000370              L6:
0077 00000370      87A7             MOVE    *FP,A7,1
0078 00000380      B7A5             MOVE    *FP(-64),A5,1
     00000390      FFC0
0079 000003A0      48A7             CMP     A5,A7
0080 000003B0      C70B             JRGT    L5
0081 000003C0                       .line   5
0082 000003C0      B7A7             MOVE    *FP(32),A7,1
     000003D0      0020
0083 000003E0      B7A5             MOVE    *FP(-32),A5,1
     000003F0      FFE0
0084 00000400      5CA7             MPYS    A5,A7
0085 00000410      B2ED             MOVE    A7,*FP(32),1
     00000420      0020
0086 00000430      87A7             MOVE    *FP,A7,1
0087 00000440      1027             ADDK    1,A7
0088 00000450      82ED             MOVE    A7,*FP,1
0089 00000460      C0F0             JRUC    L6
0090 00000470              L5:
0091 00000470                       .line   6
0092 00000470      B7A8             MOVE    *FP(32),A8,1
     00000480      0020
0093 00000490              EPI0_2:
0094 00000490      B7EE             MOVE    *SP(160),STK,1
     000004A0      00A0
0095 000004B0      09AF             MMFM    SP,A5,A7,A12,FP
     000004C0      30A0
0096 000004D0      0962             RETS    2
0097
0098                                .endfunc        15
```

Syntax **.line** *line number [, address]*

Description The .line directive creates a line number entry in the object file. Line number entries are used in symbolic debugging to associate addresses in the object code with the lines in the source code that generated them.

The .line directive has two operands:

- *Line number* indicates the line of the C source that generated a portion of code. Line numbers are relative to the beginning of the current function. This is a required parameter.
- *Address* is an expression which is the address associated with the line number. This is an optional parameter; if you don't specify an address, the assembler will use the current SPC value.

Example The .line directive is followed by the assembly language source statements that are generated by the indicated line of C source. For example, assume that the lines of C source below are line 4 and 5 in the original C source; lines 5 and 6 produce the assembly language source statements that are shown below.

C source:

```
p = 1;
for (i = 1; i <= n; ++i)
     p = p * x;
```

Resulting assembly language code:

```
0074 00000360                      .line   4
0075 00000360     838D             MOVE    A12,*FP,1
0076 00000370           L6:
0077 00000370     87A7             MOVE    *FP,A7,1
0078 00000380     B7A5             MOVE    *FP(-64),A5,1
     00000390     FFC0
0079 000003A0     48A7             CMP     A5,A7
0080 000003B0     C70B             JRGT    L5
0081 000003C0                      .line   5
0082 000003C0     B7A7             MOVE    *FP(32),A7,1
     000003D0     0020
0083 000003E0     B7A5             MOVE    *FP(-32),A5,1
     000003F0     FFE0
0084 00000400     5CA7             MPYS    A5,A7
0085 00000410     B2ED             MOVE    A7,*FP(32),1
     00000420     0020
0086 00000430     87A7             MOVE    *FP,A7,1
0087 00000440     1027             ADDK    1,A7
0088 00000450     82ED             MOVE    A7,*FP,1
0089 00000460     C0F0             JRUC    L6
0090 00000470           L5:
0091 00000470                      .line   6
0092 00000470     B7A8             MOVE    *FP(32),A8,1
     00000480     0020
```

Syntax **.member** *name, value [, type, storage class, size, tag, dims]*

Description The .member directive defines a member of a structure, union, or enumeration. It is only valid when it appears in a structure, union, or enumeration definition.

- *Name* is the name of the member that is put in the symbol table. The first 32 characters of the name are significant.
- *Value* is the value associated with the member. Any legal expression (absolute or relocatable) is acceptable.
- *Type* is the C type of the member. Appendix A contains more information about C types.
- *Storage class* is the C storage class of the member. Appendix A contains more information about C storage classes.
- *Size* is the number of bits of memory required to contain this member.
- *Tag* is the name of the type (if any) or structure of which this member is a type. This name **must** have been previously declared by a .stag, .etag, or .utag directive.
- *Dims* may be one to four expressions separated by commas. This allows up to four dimensions to be specified for the member.

The order of parameters is significant. *Name* and *value* are required parameters. All other parameters may be omitted or empty (adjacent commas indicate an empty entry). This allows you to skip a parameter and specify a parameter that occurs later in the list. Operands that are omitted or empty assume a null value.

Example Here is an example of a C structure definition and the corresponding assembly language statements:

C source:

```
struct doc
{
    char  title;
    char  group;
    int   job_number;
} doc_info;
```

Resulting assembly language code:

```
.stag   _doc,48
.member _title,0,2,8,8
.member _group,8,2,8,8
.member _job_number,16,4,8,32
.eos
```

Syntax

.stag *name [, size]*
member definitions
.eos

.etag *name [, size]*
member definitions
.eos

.utag *name [, size]*
member definitions
.eos

Description

The .stag directive begins a structure definition. The .etag directive begins an enumeration definition. The .utag directive begins a union definition. The .eos directive ends a structure, enumeration, or union definition.

- *Name* is the name of the structure, enumeration, or union. The first 32 characters of the name are significant. This is a required parameter.
- *Size* is the number of bits the structure, enumeration, or union occupies in memory. This is an optional parameter; if omitted, the size is unspecified.

The .stag, .etag, or.utag directive should be followed by a number of .member directives, which define members in the structure. The .member directive is the only directive that can appear inside a structure, enumeration, or union definition.

The assembler does not allow nested structures, enumerations, or unions. The C compiler "unwinds" nested structures by defining them separately and then referencing them from the structure they are referenced in.

Example 1

Here is an example of a structure definition.

C source:

```
struct doc
{
    char  title;
    char  group;
    int   job_number;
} doc_info;
```

Resulting assembly language code:

```
    .stag   _doc,48
    .member _title,0,2,8,8
    .member _group,8,2,8,8
    .member _job_number,16,4,8,32
    .eos
```

Example 2 Here is an example of a union definition.

C source:

```
union u_tag {
    int   val1;
    float val2;
    char  valc;
} valu;
```

Resulting assembly language code:

```
.utag   _u_tag,32
.member _val1,0,4,11,32
.member _val2,0,6,11,32
.member _valc,0,2,11,8
.eos
```

Example 3 Here is an example of an enumeration definition.

C Source:

```
{
    enum o_ty { reg_1, reg_2, result } optypes;
}
```

Resulting assembly language code:

```
.etag   _o_ty,2
.member _reg_1,0,11,16,2
.member _reg_2,1,11,16,2
.member _result,2,11,16,2
.eos
```

Syntax **.sym** *name, value [, type, storage class, size, tag, dims]*

Description The .sym directive specifies symbolic debug information about a global variable, local variable, or a function.

- *Name* is the name of the variable that is put in the object symbol table. The first 32 characters of the name are significant.
- *Value* is the value associated with the variable. Any legal expression (absolute or relocatable) is acceptable.
- *Type* is the C type of the variable. Appendix A contains more information about C types.
- *Storage class* is the C storage class of the variable. Appendix A contains more information about C storage classes.
- *Size* is the number of words of memory required to contain this variable.
- *Tag* is the name of the type (if any) or structure of which this variable is a type. This name **must** have been previously declared by a .stag, .etag, or .utag directive.
- *Dims* may be up to four expressions separated by commas. This allows up to four dimensions to be specified for the variable.

The order of parameters is significant. *Name* and *value* are required parameters. All other parameters may be omitted or empty (adjacent commas indicate an empty entry). This allows you to skip a parameter and specify a parameter that occurs later in the list. Operands that are omitted or empty assume a null value.

Example These lines of C source produce the .sym directives shown below:

C source:

```
struct s { int member1, member2; } str;
int ext;
int array[5][10];
long *ptr;
int strcmp();

main(arg1,arg2)
   int arg1;
   char *arg2;
{
   register A0;
}
```

Resulting assembly language code:

```
.sym    _str,_str,8,2,64,_s
.sym    _ext,_ext,4,2,32
.sym    _array,_array,244,2,1600,,5,10
.sym    _ptr,_ptr,21,2,32
.sym    _main,_main,36,2,0
.sym    _argl,-32,4,9,32
.sym    _arg2,-64,18,9,32
.sym    _A0,9,4,4,32
```

Appendix C

Assembler Error Messages

The assembler issues several types of error messages:

- Fatal,
- Nonfatal, **and**
- Macro errors.

When the assembler completes it second pass, it reports on any errors that it encountered during the assembly. It also prints these errors in the listing file (if one is created); an error is printed following the source line that incurred it.

This section discusses the three types of assembler error messages; they are listed in alphabetical order. Most errors are fatal errors; if an error is not fatal or if it is a macro error, this is noted in the list.

absolute value required: A relocatable symbol was used where an absolute symbol was required.

a component of the expression is invalid: Check the format of the constants in the expression.

an identifier in the expression is invalid

bad field width: The minimum field width is 0, the maximum field width is 32.

bad macro definition (macro error)

blank missing: One or more blanks must separate the fields of a source statement.

cannot open library: A library name specified with the .mlib directive does not exist or is already being used.

character constant is wider than 4 characters The maximum size of a character constant is 4 characters.

close ( ]) missing: Mismatched brackets.

close ( )) missing: Mismatched parentheses.

close quote missing: Mismatched quotes.

conflicts with previous section definition

comma missing: The assembler expected a comma but did not find one. This usually means that more operands were expected.

copy file open error: A file specified with the .copy or .include directive does not exist or is already being used.

divide by zero: An expression or well-defined expression contains invalid division.

duplicate definition: The symbol appears as an operand of a REF statements and is also used as a label, **or**, the symbol appears more than once in the label field.

$END statement missing in macro (macro error): Within the macro library, an end of file was encountered before an $END statement.

ELSE needs corresponding IF (macro error): The ELSE statement is valid only within a conditional block that is started with an IF statement.

expression not terminated properly

expression out of bounds

filename missing: This specified filename cannot be found.

illegal operation in expression

illegal structure definition

illegal structure, union, or enumeration tag

$IF level exceeded (macro error): Conditional blocks can only be nested to a maximum of 44 levels within a macro definition.

include/copy files not allowed in macros (macro error): You cannot use the .copy or .include directive within a macro.

incompatible addressing modes: An invalid combination of addressing modes has been used in an instruction.

incorrect macro definition (macro error): Within the macro library, a macro was not found or a macro name was not given for a macro call.

indirect (*) missing

invalid binary constant: The only valid binary integers are 0 and 1; the constant must be suffixed with b or B.

invalid branch displacement

invalid decimal constant: The only valid decimal integers are 0–9.

invalid expression: This may indicate invalid use of a relocatable symbol in an expression.

invalid floating point constant

invalid IF/LOOP nesting (macro error)

invalid IF structure (macro error)

invalid hexadecimal constant: The only valid hexadecimal digits are the integers 0–9 and the letters A–F. The constant must be suffixed with h or H and it must begin with an integer.

invalid macro library pathname (macro error): The macro library name that was specified with an .mlib directive is invalid.

invalid macro qualifier (macro error): The only valid macro qualifiers are S, V, L, A, SS, SV, SL, and SA.

invalid macro verb (macro error)

invalid octal constant: Valid octal digits include 0–7; the constant must be suffixed with q or Q.

invalid opcode: The command field of the source statement has an entry that is not defined as an instruction, directive, or macro.

invalid option: An option specified with the .option directive is not legal.

invalid register value

invalid symbol

label required: The flagged directive must have a label.

library not in archive format: A file specified with the .mlib directive is not an archive library.

LOOP nesting level exceeded (macro error)

long macro variable qualifier (macro error)

macro line too long (macro error)

macro nesting level exceeded (macro error)

macro variable component specified is illegal (macro error)

maximum number of copy files exceeded

open quote missing: Mismatched quotes.

operand missing

overflow in floating point constant

pass1/pass2 operand conflict

positive value required

register required

registers in opposite files

string required

symbol required

syntax error

symbol used in both ref and def

target address not word aligned

too many array dimensions

too many macro variables (macro error)

unbalanced symbol table entries

undefined symbol: An undefined symbol was used where a well-defined expression was expected.

unexpected endif encountered (macro error)

underflow in floating point constant

value truncated

variable already defined (macro error)

warning - byte value truncated: The specified value cannot be expresses in 8 bits.

warning - line truncated: The maximum line length is 200 characters.

warning - null string defined

warning - register converted to immediate

warning - SP will be corrupted

warning - string length exceeds maximum limit

warning - symbol truncated: Symbol names are significant to 32 characters.

warning - trailing operand(s)

warning - unexpected .end found in macro

warning - value out of range

warning - value truncated

Appendix D

Linker Error Messages

The linker issues several types of error messages:

- Syntax and command errors
- Allocation errors
- I/O errors

This section discusses the three types of errors; they are listed alphabetically within each category. The symbol "(...)" is used in these listings to represent the name of an object that the linker is attempting to interact with when an error occurs.

- ***Syntax/Command Errors***

  These errors are caused by incorrect use of linker directives, misuse of an input expression, invalid options, Check the syntax of all expressions, check the input directives for accuracy. Review the various options you are using and check for conflicts.

  absolute symbol (...) being redefined: An absolute symbol may not be redefined.

  adding name (...) to multiple output sections: The input section is mentioned twice in the SECTION directive.

  ALIGN illegal in this context: Alignment of a symbol may only be performed within a SECTIONS directive.

  attempt to decrement "."

  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 4-byte constant.

  binding excludes alignment: The section will be bound at the specified address regardless of the alignment of that address.

  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.

  -c requires fill value of 0 in .cinit: The value parameter has been overridden.

  -f flag does not specify a fill value

cannot align a section within GROUP - (...) not aligned

cannot bind 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.

cannot specify a page for a section within a GROUP

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.

DSECT (...) can't be linked to an attribute

-e flag does not specify a legal symbol name (...)

entry point other than _c_int00 specified: For -c option only.

entry point symbol (...) undefined

errors in input - (...) not built

fill value on -f flag truncated to (...) bytes (warning)

ifile (comfile) nesting exceeded with file (...): Command file nesting is allowed up to 16 levels.

illegal operator in expression

misuse of "." symbol in assignment instruction: The dot symbol cannot be used in assignment statements that are outside SECTIONS directives.

no input files

number (...) not a power of 2: For the ALIGN operator.

-o flag does specify a valid file name : string

option flag does not specify a number

option is invalid flag

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.

symbol referencing errors - (...) not built

symbol (...) from file (...) being redefined: A defined symbol may not be redefined in an assignment statement.

syntax error: scanned line = (...)

unexpected EOF(end of file): Syntax error in the linker command file.

undefined symbol in expression

- ***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.

binding address (...) for section (...) is outside all memory on page (...)

binding address (...) for section (...) overlays previously allocated section

binding address (...) incompatible with alignment for section (...)

can't allocate output section, (...) of size (...) on page (...)

can't allocate section (...) with attribute (...) on page (...)

default allocation failed: (...) is too large

GROUP containing section (...) is too big

internal symbol (...) redefined in file (...): Ignored.

memory types (...) and (...) on page (...) overlap

no owner (...) for section (...) on page (...): Invalid or nonexistent memory range.

output file (...) not executable (warning)

PC-relative displacement overflow at address (...) in file (...)

section (...) at address (...) overlays previously allocated section (...) at address

section (...), bound at address (...), won't fit into page (...) of configured memory

section (...) enters unconfigured memory at address (...)

section (...) in file (...) is too big

undefined symbol (...) first referenced in file (...): Unless the -r option is used, the linker requires that all referenced symbols are defined.

- *I/O Errors:*

  The following error messages indicate that the input file is corrupt, nonexistent, or unreadable or because the linker cannot write to the output file. Make sure that the input file is in the correct directory and that the file system is not out of space. If the input file is corrupt, try reassembling it.

  cannot complete output file (...), write error

  cannot create output file (...):

  can't open (...)

  can't read (...)

  can't seek (...)

  could not create map file (...)

  fail to copy (...)

  fail to read (...)

  fail to seek (...)

  fail to skip (...)

  fail to write (...)

  file (...) has no relocation information

  file (...) is of unknown type, magic number = (...)

  illegal relocation type (...) found in section(s) of file (...)

  internal error : aux table overflow

  invalid archive size for file (...)

  I/O error on output file (...)

  library (...) member has no relocation information

  line number entry found for absolute symbol

  memory allocation failure

  no symbol map produced - not enough memory

  relocation symbol not found: index (...), section (...), file (...)

  relocation entries out of order in section (...) of file (...)

  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.

  seek to (...) failed

Appendix E

ASCII Character Set

Base		Char	Base		Char	Base		Char	Base		Char
10	16		10	16		10	16		10	16	
0	00	NULL	32	20	SP	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	>
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Appendix F

Glossary

absolute address: An address that is permanently assigned to a TMS34010 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 by the .set directive.

assignment statement: A statement that assigns a value to a variable.

autoinitialization: The process of initializing global C variables (contained in the .cinit section) before beginning program execution.

auxiliary entry: A symbol may have an extra entry in the symbol table that contains additional information about the symbol (whether the symbol is a filename, as section name, a function name, etc.).

binding: A process in which you specify a distinct address for an output section or a symbol.

block: A set of declarations and statements that are grouped together with braces.

.bss: 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.

cache memory: A fast local memory onboard the TMS34010. Blocks of code that are executed repeatedly can be loaded into the cache; this reduces the number of memory cycles and speeds program execution.

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 defined for allocation. By default, all memory is configured; you can configure specific ranges of memory with the MEMORY linker directive.

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).

emulator: A hardware development system that emulates TMS34010 operation.

entry point: The starting execution point in target memory.

executable module: An object file that has been linked and can be executed in a TMS34010 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 module but defined in another module, **or**, a symbol that is defined in the current module that can be referenced by other modules.

field: For the TMS34010, a software-configurable data type whose length can be programmed to be any value in the range of 1–32 bits.

file header: A portion of a COFF object file that contains general information about the object file (such as the number of section headers, the type of system the object file can be downloaded to, the number of symbols in the symbol table, and the symbol table's starting address).

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).

GSP: TMS34010 graphics system processor.

high-level language debugging: The ability of a compiler to retain symbolic and high-level language information (such as type and function definitions) so that a debugging tool can use this information.

hole: An area between the input sections that 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.

line number entry: An entry in a COFF output module that maps lines of assembly code back to the original C source file that created them.

linker: A software tool that combines object files to form an object module that can be allocated into TMS34010 system memory and executed by the TMS34010.

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 TMS34010 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.

magic number: An entry in the COFF file header that identifies an object file as a module that can be executed by the TMS34010.

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.

member: (1) An element or variable of a structure, union, or enumeration. (2) An individual file within an archive library.

memory map: A map of TMS34010 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: A section that is defined with a .sect or .usect directive. The .sect directive defines *initialized* named sections that can be used like the .text and .data sections. The .usect directive defines *uninitialized* named sections that can be used like the .bss section.

object file: A file that has been assembled or linked and contains machine-language object code.

object format converter: A program that converts a COFF object file into an Intel, Tektronix, or TI-tagged format object file.

object library: An archive library made up of individual object files.

operand: The arguments, or parameters, of an assembly language instruction, assembler directive, or macro directive.

optional header: A portion of a COFF object file that the linker uses to perform relocation at download time.

options: Command parameters that allow you to request additional or specific functions when you invoke a software tool.

output module: A linked, executable object file that can be downloaded and executed on a target system.

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. A TMS34010 system 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.

RAM model: An autoinitialization model used by the linker when linking C code. The linker uses this model when you invoke the linker with the -cr option. The RAM model allows variables to be initialized at load time instead of run time.

raw data: Executable code or initialized data in an output section.

relocation: A process in which the linker adjusts all the references to a symbol when the symbol's address changes.

ROM model: An autoinitialization model used by the linker when linking C code. The linker uses this model when you invoke the linker with the -c option. The ROM model tells the linker to load the .cinit section of data tables into memory; variables are initialized at run time.

section: A relocatable block of code or data that will ultimately occupy contiguous space in the TMS34010 memory map.

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.

sign-extend: Fill unused MSBs of a value with the value's sign bit.

simulator: A software development system that simulates TMS34010 operation.

source file: A file that contains C code or TMS34010 assembly language code that will be compiled or assembled to form an object file.

SPC: section program counter

static: Refers to a variable whose scope is confined to a function or a program. The values of static variables are not discarded when the function or program is exited; their previous values are resumed when the function or program is re-entered.

storage class: Any entry in the symbol table that indicates how a symbol should be accessed.

string table: 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.

structure: A collection of one or more variables grouped together under a single name.

symbol: A string of alphanumeric characters that represents an address or a value.

symbolic debugging: The ability of a software tool to retain symbolic information so that it can be used by a debugging tool such as a simulator or an emulator.

symbol table: A portion of a COFF object file that contains information about the symbols that are defined and used by the file.

tag: An optional "type" name that can be assigned to a structure, union, or enumeration.

target memory: Physical memory in a TMS34010-based system into which executable object code will be loaded.

.text: 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 TMS34010 memory map and cannot be loaded with code or data.

uninitialized section: A COFF section that reserves space in the TMS34010 memory map but that has no actual contents. These sections are built up with the .bss and .usect directives.

union: A variable which may hold (at different times) object of different types and sizes.

unsigned: Refers to a value that is treated as a positive number, regardless of its actual sign.

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 32-bit addressable location in target memory.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

TI Sales Offices

ALABAMA: Huntsville (205) 837-7530

ARIZONA: Phoenix (602) 995-1007;
Tucson (602) 624-3276

CALIFORNIA: Irvine (714) 660-8187;
Sacramento (916) 929-0192;
San Diego (619) 278-9601;
Santa Clara (408) 980-9000;
Torrance (213) 217-7010;
Woodland Hills (818) 704-7759

COLORADO: Aurora (303) 368-8000

CONNECTICUT: Wallingford (203) 269-0074

FLORIDA: Ft. Lauderdale (305) 973-8502;
Altamonte Springs (305) 260-2116;
Tampa (813) 870-6420

GEORGIA: Norcross (404) 662-7900

ILLINOIS: Arlington Heights (312) 640-2925

INDIANA: Ft. Wayne (219) 424-5174;
Carmel (317) 573-6400

IOWA: Cedar Rapids (319) 395-9550

MARYLAND: Baltimore (301) 944-8600

MASSACHUSETTS: Waltham (617) 895-9100

MICHIGAN: Farmington Hills (313) 553-1500;
Grand Rapids (616) 957-4200

MINNESOTA: Eden Prairie (612) 828-9300

MISSOURI: Kansas City (816) 523-2500;
St. Louis (314) 569-7600

NEW JERSEY: Iselin (201) 750-1050

NEW MEXICO: Albuquerque (505) 345-2555

NEW YORK: East Syracuse (315) 463-9291;
Melville (516) 454-6600; **Pittsford** (716) 385-6770;
Poughkeepsie (914) 473-2900

NORTH CAROLINA: Charlotte (704) 527-0930;
Raleigh (919) 876-2725

OHIO: Beachwood (216) 464-6100;
Dayton (513) 258-3877

OREGON: Beaverton (503) 643-6758

PENNSYLVANIA: Blue Bell (215) 825-9500

PUERTO RICO: Hato Rey (809) 753-8700

TENNESSEE: Johnson City (615) 461-2192

TEXAS: Austin (512) 250-7655;
Houston (713) 778-6592; **Richardson** (214) 680-5082;
San Antonio (512) 496-1779

UTAH: Murray (801) 266-8972

VIRGINIA: Fairfax (703) 849-1400

WASHINGTON: Redmond (206) 881-3080

WISCONSIN: Brookfield (414) 785-7140

CANADA: Nepean, Ontario (613) 726-1970;
Richmond Hill, Ontario (416) 884-9181;
St. Laurent, Quebec (514) 336-1860

TI Regional Technology Centers

CALIFORNIA: Irvine (714) 660-8140;
Santa Clara (408) 748-2220

GEORGIA: Norcross (404) 662-7945

ILLINOIS: Arlington Heights (312) 640-2909

MASSACHUSETTS: Waltham (617) 895-9197

TEXAS: Richardson (214) 680-5066

CANADA: Nepean, Ontario (613) 726-1970

Customer Response Center

TOLL FREE: (800) 232-3200

OUTSIDE USA: (214) 995-6611
(8:00 a.m. — 5:00 p.m. CST)

TI Distributors

TI AUTHORIZED DISTRIBUTORS
Arrow Electronics (U.S. and Canada)
Future Electronics (Canada)
General Radio Supply Company
Hall-Mark Electronics
Kierulff Electronics
Marshall Industries
Newark Electronics
Schweber Electronics
Time Electronics
Wyle Laboratories
Zeus Components

—OBSOLETE PRODUCT ONLY—
Rochester Electronics, Inc.
Newburyport, Massachusetts
(617) 462-9332

ALABAMA: Arrow (205) 837-6955;
Hall-Mark (205) 837-8700; Kierulff (205) 883-6070;
Marshall (205) 881-9235; Schweber (205) 895-0480

ARIZONA: Arrow (602) 968-4800;
Hall-Mark (602) 437-1200; Kierulff (602) 437-0750;
Marshall (602) 968-6181; Schweber (602) 997-4874;
Wyle (602) 866-2888

CALIFORNIA: Los Angeles/Orange County:
Arrow (818) 701-7500; (714) 838-5422;
Hall-Mark (818) 716-7300, (714) 669-4700,
(213) 217-8400; Kierulff (213) 725-0325, (714) 731-5711,
(714) 220-6300; (818) 407-2500;
Marshall (818) 407-0101, (818) 459-5500,
(714) 458-5395; Schweber (818) 999-4702,
(714) 863-0200; (213) 327-8409; Wyle (213) 322-9953;
(818) 880-9000, (714) 863-9953; Zeus (714) 921-9000;
Sacramento: Hall-Mark (916) 722-8600;
Marshall (916) 635-9700; Schweber (916) 929-9732;
Wyle (916) 638-5282;
San Diego: Arrow (619) 565-4800;
Hall-Mark (619) 268-1201; Kierulff (619) 278-2112;
Marshall (619) 578-9600; Schweber (619) 450-0454;
Wyle (619) 565-9171;
San Francisco Bay Area: Arrow (408) 745-6600,
(415) 487-4600; Hall-Mark (408) 432-0900;
Kierulff (408) 971-2600; Marshall (408) 942-4600;
Schweber (408) 432-7171; Wyle (408) 727-2500;
Zeus (408) 998-5121

COLORADO: Arrow (303) 696-1111;
Hall-Mark (303) 790-1662; Kierulff (303) 790-4444;
Marshall (303) 451-8444; Schweber (303) 799-0258;
Wyle (303) 457-9953

CONNECTICUT: Arrow (203) 265-7741;
Hall-Mark (203) 269-0100; Kierulff (203) 265-1115;
Marshall (203) 265-3822; Schweber (203) 748-7080

FLORIDA: Ft. Lauderdale: Arrow (305) 429-8200;
Hall-Mark (305) 971-9280; Kierulff (305) 486-4004;
Marshall (305) 977-4880; Schweber (305) 977-7511;
Orlando: Arrow (305) 725-1480;
Hall-Mark (305) 855-4020; Kierulff (305) 682-6923;
Marshall (305) 841-1878; Schweber (305) 331-7555;
Zeus (305) 365-3000;
Tampa: Hall-Mark (813) 530-4543;
Marshall (813) 576-1399

GEORGIA: Arrow (404) 449-8252;
Hall-Mark (404) 447-8000; Kierulff (404) 447-5252;
Marshall (404) 923-5750; Schweber (404) 449-9170

ILLINOIS: Arrow (312) 397-3440;
Hall-Mark (312) 860-3800; Kierulff (312) 250-0500;
Marshall (312) 490-0155; Newark (312) 784-5100;
Schweber (312) 364-3750

INDIANA: Indianapolis: Arrow (317) 243-9353;
Hall-Mark (317) 872-8875; Marshall (317) 297-0483

IOWA: Arrow (319) 395-7230;
Schweber (319) 373-1417

KANSAS: Kansas City: Arrow (913) 541-9542;
Hall-Mark (913) 888-4747; Marshall (913) 492-3121;
Schweber (913) 492-2921

MARYLAND: Arrow (301) 995-0003;
Hall-Mark (301) 988-9800; Kierulff (301) 840-1155;
Marshall (301) 840-9450; Schweber (301) 840-5900;
Zeus (301) 997-1118

MASSACHUSETTS: Arrow (617) 933-8130;
Hall-Mark (617) 667-0902; Kierulff (617) 667-8331;
Marshall (617) 658-0810; Schweber (617) 275-5100,
(617) 657-0760; Time (617) 532-6200;
Zeus (617) 863-8800

MICHIGAN: Detroit: Arrow (313) 971-8220;
Marshall (313) 525-5850; Newark (313) 967-0600;
Schweber (313) 525-8100;
Grand Rapids: Arrow (616) 243-0912

MINNESOTA: Arrow (612) 830-1800;
Hall-Mark (612) 941-2600; Kierulff (612) 941-7500;
Marshall (612) 559-2211; Schweber (612) 941-5280

MISSOURI: St. Louis: Arrow (314) 567-6888;
Hall-Mark (314) 291-5350; Kierulff (314) 997-4956;
Schweber (314) 739-0526

NEW HAMPSHIRE: Arrow (603) 668-6968;
Schweber (603) 625-2250

NEW JERSEY: Arrow (201) 575-5300,
(609) 596-8000; General Radio (609) 964-8560;
Hall-Mark (201) 575-4415, (609) 235-1900;
Kierulff (201) 575-6750, (609) 235-1444;
Marshall (201) 882-0320, (609) 234-9100;
Schweber (201) 227-7880

NEW MEXICO: Arrow (505) 243-4566

NEW YORK: Long Island: Arrow (516) 231-1000;
Hall-Mark (516) 737-0600; Marshall (516) 273-2053;
Schweber (516) 334-7555; Zeus (914) 937-7400
Rochester: Arrow (716) 427-0300;
Hall-Mark (716) 244-9290; Marshall (716) 235-7620;
Schweber (716) 424-2222
Syracuse: Marshall (607) 798-1611

NORTH CAROLINA: Arrow (919) 876-3132,
(919) 725-8711; Hall-Mark (919) 872-0712;
Kierulff (919) 872-8410; Marshall (919) 878-9882;
Schweber (919) 876-0000

OHIO: Cleveland: Arrow (216) 248-3990;
Hall-Mark (216) 349-4632; Kierulff (216) 831-5222;
Marshall (216) 248-1788; Schweber (216) 464-2970
Columbus: Arrow (614) 885-8362;
Hall-Mark (614) 888-3313;
Dayton: Arrow (513) 435-5563;
Kierulff (513) 439-0045; Marshall (513) 236-8088;
Schweber (513) 439-1800

OKLAHOMA: Arrow (918) 665-7700;
Kierulff (918) 252-7537; Schweber (918) 622-8000

OREGON: Arrow (503) 684-1690;
Kierulff (503) 641-9153; Wyle (503) 640-6000;
Marshall (503) 644-5050

PENNSYLVANIA: Arrow (412) 856-7000,
(215) 928-1800; General Radio (215) 922-7037;
Schweber (215) 441-0600, (412) 782-1600

TEXAS: Austin: Arrow (512) 835-4180;
Hall-Mark (512) 258-8848; Kierulff (512) 835-2090;
Marshall (512) 837-1991; Schweber (512) 458-8253;
Wyle (512) 834-9957;
Dallas: Arrow (214) 380-6464;
Hall-Mark (214) 553-4300; Kierulff (214) 840-0110;
Marshall (214) 233-5200; Schweber (214) 661-5010;
Wyle (214) 235-9953; Zeus (214) 783-7010;
Houston: Arrow (713) 530-4700;
Hall-Mark (713) 781-6100; Kierulff (713) 530-7030;
Marshall (713) 895-9200; Schweber (713) 784-3600;
Wyle (713) 879-9953

UTAH: Arrow (801) 972-0404;
Hall-Mark (801) 972-1008; Kierulff (801) 973-6913;
Marshall (801) 485-1551; Wyle (801) 974-9953

WASHINGTON: Arrow (206) 643-4800;
Kierulff (206) 575-4420; Wyle (206) 453-8300;
Marshall (206) 747-9100

WISCONSIN: Arrow (414) 792-0150;
Hall-Mark (414) 797-7844; Kierulff (414) 784-8160;
Marshall (414) 797-8400; Schweber (414) 784-9020

CANADA: Calgary: Future (403) 235-5325;
Edmonton: Future (403) 438-2858;
Montreal: Arrow Canada (514) 735-5511;
Future (514) 694-7710;
Ottawa: Arrow Canada (613) 226-6903;
Future (613) 820-8313;
Quebec City: Arrow Canada (418) 687-4231;
Toronto: Arrow Canada (416) 672-7769;
Future (416) 638-4771;
Vancouver: Future (604) 294-1166;
Winnipeg: Future (204) 339-0554

TEXAS INSTRUMENTS

BT

TI Worldwide Sales Offices

ALABAMA: Huntsville: 500 Wynn Drive, Suite 514, Huntsville, AL 35805, (205) 837-7530.

ARIZONA: Phoenix: 8825 N. 23rd Ave., Phoenix, AZ 85021, (602) 995-1007.

CALIFORNIA: Irvine: 17891 Cartwright Rd., Irvine, CA 92714, (714) 660-8187; **Sacramento:** 1900 Point West Way, Suite 171, Sacramento, CA 95815, (916) 929-1521; **San Diego:** 4333 View Ridge Ave., Suite B, San Diego, CA 92123, (619) 278-9601; **Santa Clara:** 5353 Betsy Ross Dr., Santa Clara, CA 95054, (408) 980-9000; **Torrance:** 690 Knox St., Torrance, CA 90502, (213) 217-7010; **Woodland Hills:** 21220 Erwin St., Woodland Hills, CA 91367, (818) 704-7759.

COLORADO: Aurora: 1400 S. Potomac Ave., Suite 101, Aurora, CO 80012, (303) 368-8000.

CONNECTICUT: Wallingford: 9 Barnes Industrial Park Rd., Barnes Industrial Park, Wallingford, CT 06492, (203) 269-0074.

FLORIDA: Ft. Lauderdale: 2765 N.W. 62nd St., Ft. Lauderdale, FL 33309, (305) 973-8502; **Maitland:** 2601 Maitland Center Parkway, Maitland, FL 32751, (305) 660-4600; **Tampa:** 5010 W. Kennedy Blvd., Suite 101, Tampa, FL 33609, (813) 870-6420.

GEORGIA: Norcross: 5515 Spalding Drive, Norcross, GA 30092, (404) 662-7900.

ILLINOIS: Arlington Heights: 515 W. Algonquin, Arlington Heights, IL 60005, (312) 640-2925.

INDIANA: Ft. Wayne: 2020 Inwood Dr., Ft. Wayne, IN 46815, (219) 424-5174; **Indianapolis:** 2346 S. Lynhurst, Suite J-400, Indianapolis, IN 46241, (317) 248-8555.

IOWA: Cedar Rapids: 373 Collins Rd. NE, Suite 200, Cedar Rapids, IA 52402, (319) 395-9550.

MARYLAND: Baltimore: 1 Rutherford Pl., 7133 Rutherford Rd., Baltimore, MD 21207, (301) 944-8600.

MASSACHUSETTS: Waltham: 504 Totten Pond Rd., Waltham, MA 02154, (617) 895-9100.

MICHIGAN: Farmington Hills: 33737 W. 12 Mile Rd., Farmington Hills, MI 48018, (313) 553-1500.

MINNESOTA: Eden Prairie: 11000 W. 78th St., Eden Prairie, MN 55344 (612) 828-9300.

MISSOURI: Kansas City: 8080 Ward Pkwy., Kansas City, MO 64114, (816) 523-2500; **St. Louis:** 11816 Borman Drive, St. Louis, MO 63146, (314) 569-7600.

NEW JERSEY: Iselin: 485E U.S. Route 1 South, Parkway Towers, Iselin, NJ 08830 (201) 750-1050

NEW MEXICO: Albuquerque: 2820-D Broadbent Pkwy NE, Albuquerque, NM 87107, (505) 345-2555.

NEW YORK: East Syracuse: 6365 Collamer Dr., East Syracuse, NY 13057, (315) 463-9291; **Endicott:** 112 Nanticoke Ave., P.O. Box 618, Endicott, NY 13760, (607) 754-3900; **Melville:** 1 Huntington Quadrangle, Suite 3C10, P.O. Box 2936, Melville, NY 11747, (516) 454-6600; **Pittsford:** 2851 Clover St., Pittsford, NY 14534, (716) 385-6770; **Poughkeepsie:** 385 South Rd., Poughkeepsie, NY 12601, (914) 473-2900.

NORTH CAROLINA: Charlotte: 8 Woodlawn Green, Woodlawn Rd., Charlotte, NC 28210, (704) 527-0930; **Raleigh:** 2809 Highwoods Blvd., Suite 100, Raleigh, NC 27625, (919) 876-2725.

OHIO: Beachwood: 23408 Commerce Park Rd., Beachwood, OH 44122, (216) 464-6100; **Dayton:** Kingsley Bldg., 4124 Linden Ave., Dayton, OH 45432, (513) 258-3877.

OREGON: Beaverton: 6700 SW 105th St., Suite 110, Beaverton, OR 97005, (503) 643-6758.

PENNSYLVANIA: Ft. Washington: 260 New York Dr., Ft. Washington, PA 19034, (215) 643-6450; **Coraopolis:** 420 Rouser Rd., 3 Airport Office Park, Coraopolis, PA 15108, (412) 771-8550.

PUERTO RICO: Hato Rey: Mercantil Plaza Bldg., Suite 505, Hato Rey, PR 00919, (809) 753-8700.

TEXAS: Austin: P.O. Box 2909, Austin, TX 78769, (512) 250-7655; **Richardson:** 1001 E. Campbell Rd., Richardson, TX 75080, (214) 680-5082; **Houston:** 9100 Southwest Frwy., Suite 237, Houston, TX 77036, (713) 778-6592; **San Antonio:** 1000 Central Parkway South, San Antonio, TX 78232, (512) 496-1779.

UTAH: Murray: 5201 South Green SE, Suite 200, Murray, UT 84107, (801) 266-8972.

VIRGINIA: Fairfax: 2750 Prosperity, Fairfax, VA 22031, (703) 849-1400.

WASHINGTON: Redmond: 5010 148th NE, Bldg B, Suite 107, Redmond, WA 98052, (206) 881-3080.

WISCONSIN: Brookfield: 450 N. Sunny Slope, Suite 150, Brookfield, WI 53005, (414) 785-7140.

CANADA: Nepean: 301 Moodie Drive, Mallorn Center, Nepean, Ontario, Canada, K2H9C4, (613) 726-1970. **Richmond Hill:** 280 Centre St. E., Richmond Hill L4C1B1, Ontario, Canada (416) 884-9181; **St. Laurent:** Ville St. Laurent Quebec, 9460 Trans Canada Hwy., St. Laurent, Quebec, Canada H4S1R7, (514) 335-8392.

ARGENTINA: Texas Instruments Argentina S.A.I.C.F.: Esmeralda 130, 15th Floor, 1035 Buenos Aires, Argentina, 1 + 394-3008.

AUSTRALIA (& NEW ZEALAND): Texas Instruments Australia Ltd.: 6-10 Talavera Rd., North Ryde (Sydney), New South Wales, Australia 2113, 2 + 887-1122; 5th Floor, 418 St. Kilda Road, Melbourne, Victoria, Australia 3004, 3 + 267-4677; 171 Philip Highway, Elizabeth, South Australia 5112, 8 + 255-2066.

AUSTRIA: Texas Instruments Ges.m.b.H.: Industriestrabe B/16, A-2345 Brunn/Gebirge, 2236-846210.

BELGIUM: Texas Instruments N.V. Belgium S.A.: Mercure Centre, Raketstraat 100, Rue de la Fusee, 1130 Brussels, Belgium, 2/720.80.00.

BRAZIL: Texas Instruments Electronicos do Brasil Ltda.: Rua Paes Leme, 524-7 Andar Pinheiros, 05424 Sao Paulo, Brazil, 0815-6166.

DENMARK: Texas Instruments A/S, Mairelundvej 46E, DK-2730 Herlev, Denmark, 2 - 91 74 00.

FINLAND: Texas Instruments Finland OY: Teollisuuskatu 19D 00511 Helsinki 51, Finland, (90) 701-3133.

FRANCE: Texas Instruments France: Headquarters and Prod. Plant, BP 05, 06270 Villeneuve-Loubet, (93) 20-01-01; Paris Office, BP 67 8-10 Avenue Morane-Saulnier, 78141 Velizy-Villacoublay, (3) 946-97-12; Lyon Sales Office, L'Oree D'Ecully, Batiment B, Chemin de la Forestiere, 69130 Ecully, (7) 833-04-40; Strasbourg Sales Office, Le Sebastopol 3, Quai Kleber, 67055 Strasbourg Cedex, (88) 22-12-66; Rennes, 23-25 Rue du Puits Mauger, 35100 Rennes, (99) 31-54-86; Toulouse Sales Office, Le Peripole—2, Chemin du Pigeonnier de la Cepiere, 31100 Toulouse, (61) 44-18-19; Marseille Sales Office, Noilly Paradis—146 Rue Paradis, 13006 Marseille, (91) 37-25-30.

TEXAS INSTRUMENTS

GERMANY (Fed. Republic of Germany): Texas Instruments Deutschland GmbH: Haggertystrasse 1, D-8050 Freising, 8161 + 80-4591; Kurfuerstendamm 195/196, D-1000 Berlin 15, 30 + 882-7365; III, Hagen 43/Kibbelstrasse, 19, D-4300 Essen, 201-24250; Frankfurter Allee 6-8, D-6236 Eschborn 1, 06196 + 8070; Hamburgerstrasse 11, D-2000 Hamburg 76, 040 + 220-1154, Kirchhorsterstrasse 2, D-3000 Hannover 51, 511 + 648021; Maybachstrabe 11, D-7302 Ostfildern 2-Nelingen, 711 + 547001; Mixikoring 19, D-2000 Hamburg 60, 40 + 637 + 0061; Postfach 1309, Roonstrasse 16, D-5400 Koblenz, 261 + 35044.

HONG KONG (+ PEOPLES REPUBLIC OF CHINA): Texas Instruments Asia Ltd., 8th Floor, World Shipping Ctr., Harbour City, 7 Canton Rd., Kowloon, Hong Kong, 3 + 722-1223.

IRELAND: Texas Instruments (Ireland) Limited: Brewery Rd., Stillorgan, County Dublin, Eire, 1 831311.

ITALY: Texas Instruments Semicondutttori Italia Spa: Viale Delle Scienze, 1, 02015 Cittaducale (Rieti), Italy, 746 694.1; Via Salaria KM 24 (Palazzo Cosma), Monterotondo Scalo (Rome), Italy, 6 + 9003241; Viale Europa, 38-44, 20093 Cologno Monzese (Milano), 2 2532541; Corso Svizzera, 185, 10100 Torino, Italy, 11 774545; Via J. Barozzi 6, 40100 Bologna, Italy, 51 355851.

JAPAN: Texas Instruments Asia Ltd.: 4F Aoyama Fuji Bldg., 6-12, Kita Aoyama 3-Chome, Minato-ku, Tokyo, Japan 107, 3-498-2111; Osaka Branch, 5F, Nissho Iwai Bldg., 30 Imabashi 3- Chome, Higashi-ku, Osaka, Japan 541, 06-204-1881; Nagoya Branch, 7F Daini Toyota West Bldg., 10-27, Meieki 4-Chome, Nakamura-ku Nagoya, Japan 450, 52-583-8691.

KOREA: Texas Instruments Supply Co.: 3rd Floor, Samon Bldg., Yuksam-Dong, Gangnam-ku, 135 Seoul, Korea, 2 + 462-8001.

MEXICO: Texas Instruments de Mexico S.A.: Mexico City, AV Reforma No. 450 — 10th Floor, Mexico, D.F., 06600, 5 + 514-3003.

MIDDLE EAST: Texas Instruments: No. 13, 1st Floor Mannai Bldg., Diplomatic Area, P.O. Box 26335, Manama Bahrain, Arabian Gulf, 973 + 274681.

NETHERLANDS: Texas Instruments Holland B.V., P.O. Box 12995, (Bullewijk) 1100 CB Amsterdam, Zuid-Oost, Holland 20 + 5602911.

NORWAY: Texas Instruments Norway A/S: PB106, Refstad 131, Oslo 1, Norway, (2) 155090.

PHILIPPINES: Texas Instruments Asia Ltd.: 14th Floor, Ba- Lepanto Bldg., 8747 Paseo de Roxas, Makati, Metro Manila, Philippines, 2 + 8188987.

PORTUGAL: Texas Instruments Equipamento Electronico (Portugal), Lda.: Rua Eng. Frederico Ulrich, 2650 Moreira Da Maia, 4470 Maia, Portugal, 2-948-1003.

SINGAPORE (+ INDIA, INDONESIA, MALAYSIA, THAILAND): Texas Instruments Asia Ltd.: 12 Lorong Bakar Batu, Unit 01-02, Kolam Ayer Industrial Estate, Republic of Singapore, 747-2255.

SPAIN: Texas Instruments Espana, S.A.: C/Jose Lazaro Galdiano No. 6, Madrid 16, 1/458.14.58.

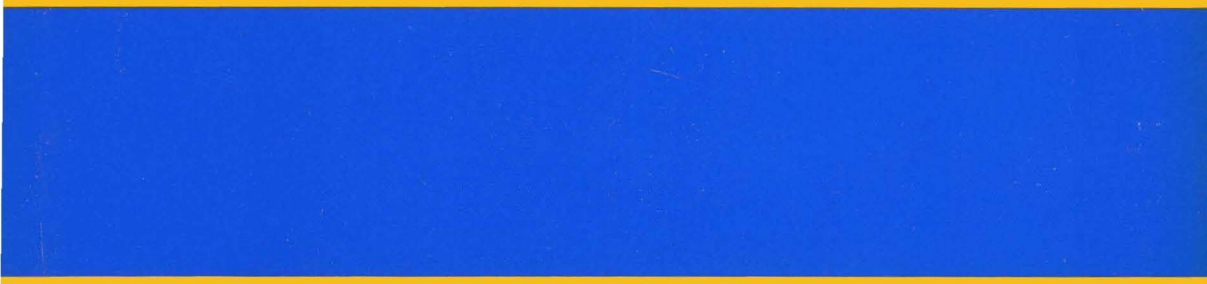
SWEDEN: Texas Instruments International Trade Corporation (Sverigefiilialen): Box 39103, 10054 Stockholm, Sweden, 8 - 235480.

SWITZERLAND: Texas Instruments, Inc., Reidstrasse 6, CH-8953 Dietikon (Zuerich) Switzerland, 1-740 2220.

TAIWAN: Texas Instruments Supply Co.: Room 903, 205 Tun Hwan Rd., 71 Sung-Kiang Road, Taipei, Taiwan, Republic of China, 2 + 521-9321.

UNITED KINGDOM: Texas Instruments Limited: Manton Lane, Bedford, MK41 7PA, England, 0234 67466; St. James House, Wellington Road North, Stockport, SK4 2RT, England, 61 + 442-7162.

BM



TEXAS INSTRUMENTS

Printed in U.S.A.
1604897-9704

SPVU004A