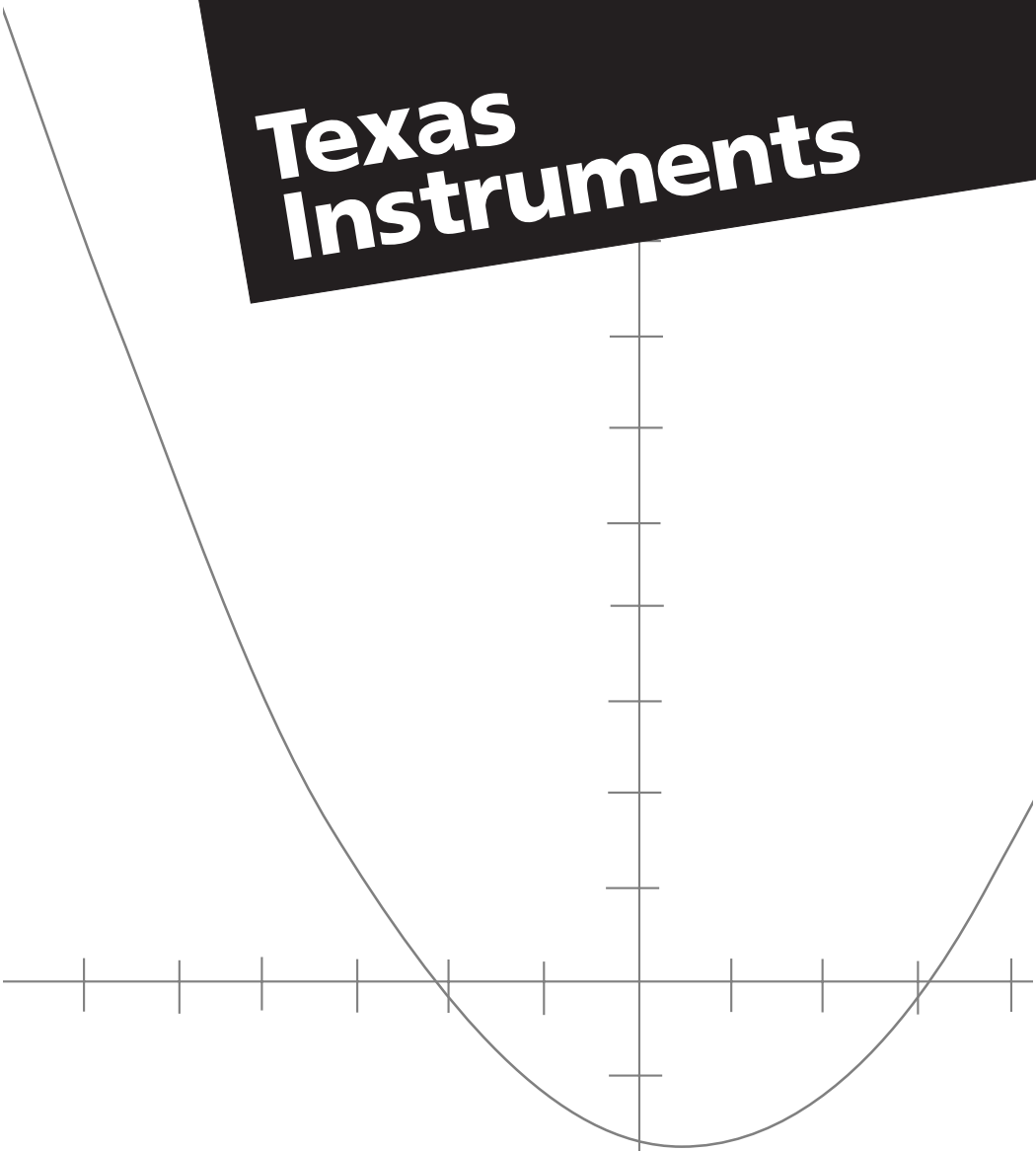
Texas Instruments



TI-89 / TI-92 Plus

Sierra C™ Assembler Reference Manual, Beta Version .02

Important information

Texas Instruments makes no warranty, either expressed or implied, including but not limited to any implied warranties of merchantability and fitness for a particular purpose, regarding any programs or book materials and makes such materials available solely on an “as-is” basis.

In no event shall Texas Instruments be liable to anyone for special, collateral, incidental, or consequential damages in connection with or arising out of the purchase or use of these materials, and the sole and exclusive liability of Texas Instruments, regardless of the form of action, shall not exceed the purchase price of this product. Moreover, Texas Instruments shall not be liable for any claim of any kind whatsoever against the use of these materials by any other party.

The latest version of this Guide, along with all other up-to-date information for developers, is available at www.ti.com/calc/developers/.

Table of Contents

Figures

Tables

Section 1: General Information

Figures

Tables

1. General Information

1.1. Introduction

This manual describes the Sierra tools, including the compiler, assembler, and linker invoked by the TI **FLASH** Studio™ for development of Flash applications (apps) and Assembly Language Programs (ASMs) for the TI-89 / TI-92 Plus calculators, and other Plusutilities that are available to the developer. They were developed by Sierra Systems to support certain Motorola processors and coprocessors and IEEE format floating-point numbers. Under license from Sierra Systems, Texas Instruments has modified this software to support TI BCD floating-point numbers, and support for coprocessors has been removed. Although the software has not been modified to exclude support for processors other than the 68000, the 68000 is the only processor supported by Texas Instruments. The license from Texas Instruments to use these products is restricted to development of software that is targeted to execute only on TI calculators.

Typically, the TI **FLASH** Studio will handle all invocations of the compiler, assembler, and linker but information is included in the various sections to enable developers to use them directly from the command line or create their own makefile if they wish, although this is not encouraged.

Section 1 contains information that applies to all the tools and describes the format of the object file generated.

Section 2 discusses features of the Sierra Systems C compiler, **com68**. These include number formats, function calls, integer and floating-point arithmetic, sections, register allocation, macros, possible error messages, and many others.

Section 3 describes both Sierra Systems assemblers: **asm68** which is invoked by the TI **FLASH** Studio, and **asm68k** which is included for developers who may wish to take advantage of the macro support it provides. Explanations of assembler syntax, symbols, constants, expression evaluation, addressing modes, and complete descriptions of assembler directives for both assemblers and the **asm68k** macros are in this section.

Section 4 is a brief discussion of the Sierra Systems linker, **link68**, and some of its features including relocation hole compression.

Section 5 describes the Sierra Systems utilities **nm68** and **size68**, provided as part of the TI-89 / TI-92 Plus SDK for use by developers.

Conventions used throughout this manual include:

- **Bold** text is used for names of functions, routines, files, keywords, directives, macros, flags, and registers. It is also used occasionally for emphasis.
- The `Courier` font is used to distinguish assembly or C program text.
- *Italicized* text indicates an input parameter that should be replaced by actual data when used in code or entered on the command line.
- Brackets ([ ]) enclose optional items.
- The vertical bar (|) indicates that the separated items are alternatives.

1.2. Command Line Wildcard Expansion

A number of Sierra C™ utilities expand wildcards in command line filenames.

Wildcard characters are defined as follows:

`*`	Match zero or more characters, where characters matched may be any character except the period (`.`). Dot-star (`.*`) at the end of a wildcard string matches files with and without extensions.
?	Match any character except the period (`.`).
`[`*`char_set`*`]`	Match any character in the character set *char_set*. Characters can be listed individually or as members of a range. A range is denoted by a hyphen (`-`) separated pair of characters; it includes the two characters and all characters lexically between them.

The following are examples of wildcard expansions:

`*.*`	Match all filenames in the current directory.
`*`	Match all filenames without extensions in the current directory.
`*.?*`	Match all filenames with extensions in the current directory.
`*.c *.s`	Match all filenames with extensions of **.c** or **.s** in the current directory.
`*.[cs]`	Same as ***.c *.s**
`c:\doc\version[0-9a-f].doc`	Match all filenames in the subdirectory **c:\doc** with a base name of version followed by a hexadecimal digit and a **.doc** extension.

The following Sierra C utilities support command line wildcard expansion:

nm68 Symbol Table Name Utility

size68 Object File Size Utility

1.3. Environment Variables

Some Sierra C utilities need to know the location of certain standard directories to locate various files. For example, the compiler must know where to search for files specified by the **#include** *filename* preprocessor directive. Since this information varies depending on the installation, the utilities examine environment variables to obtain information on the locations of these standard directories.

Figure 1.1 shows the default standard include (**include**), standard library (**lib**), executable (**bin**), and temporary (**tmp**) directories that are created by appending subdirectory names to the directory specified by the **SIERRA** environment variable. The **SIERRA** environment variable will be set when the setup program for the TI **FLASH** Studio is executed. The **SIERRA** environment variable must be set before using any of the Sierra utilities.

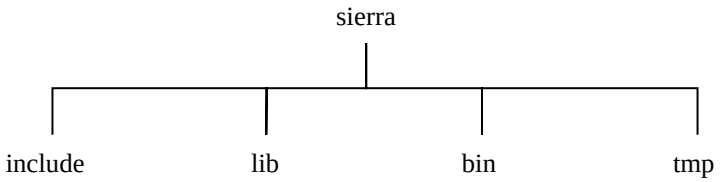


Figure 1.1: Partial Sierra C Directory Structure

Table 1.1 lists the environment variables referenced by Sierra C utilities together with the utilities that reference the variables and the files that are referenced.

Variable	Utility	Files referenced
INCLUDE68	**com68**	standard include files
LIB68	**link68**	standard libraries
	link68	default configuration file

Table 1.1: Environment Variables

Depending on which environment variables are set, the utilities that generate temporary files will put their files in one of four places. The three environment variables **TMP68**, **TMP**, and **SIERRA** are searched for in the order listed with the search terminating after the first defined variable is located. If either **TMP68** or **TMP** is defined, the directory specified by that environment variable is used to hold temporary files. If the **SIERRA** environment variable is the only variable

defined, the **tmp** subdirectory located under the directory specified by **SIERRA** is used to hold temporary files. If none of the three environment variables is specified, the current directory is used to hold temporary files.

The C preprocessor searches for the two environment variables **INCLUDE68** and **SIERRA** in the order listed to determine the locations of the standard include directories. If **INCLUDE68** is defined, the semicolon-separated list of one or more directory paths defines the standard include directories. If the **SIERRA** environment variable is the only variable defined, the **include** subdirectory located under the directory specified by **SIERRA** is used as the standard include directory.

Utilities that search for files in standard include directories search for the two environment variables **LIB68** and **SIERRA** in the order listed to determine the locations of the standard include directories. If **LIB68** is defined, the semicolon-separated list of one or more directory paths defines the standard library directories. If the **SIERRA** environment variable is the only variable defined, the **lib** subdirectory located under the directory specified by **SIERRA** is used as the standard library directory.

1.4. Object File Format

This section describes the Common Object File Format (COFF) used by TI **FLASH** Studio. COFF is the format of the object files created by the assemblers and linker, and recognized by TI **FLASH** Studio and various object file examination utilities. COFF is a standard file format and the Sierra Systems implementation (with extensions disabled, **-c** assembler command line flag) completely conforms to the format recognized by the AT&T UNIX System V operating system.

COFF is ideally suited for developing embedded applications. Space is provided for symbol and line number information used by debuggers and other applications. Executable files can be divided into numerous independent sections to allow support for systems with highly fragmented address spaces.

An object file contains the following:

- A file header
- Optional header information
- A table of section headers
- Data corresponding to the section header
- Relocation information
- Line numbers
- A symbol table
- A string table

Table 1.2 shows the overall object file structure.

FILE HEADER
Optional Information
Section 1 Header
. . .
Section n Header
Raw Data for Section 1
. . .
Raw Data for Section n
Relocation Info. for Section 1
. . .
Relocation Info. for Section n
Line Numbers for Section 1
. . .
Line Numbers for Section n
SYMBOL TABLE
STRING TABLE

Table 1.2: Object File Layout

Some or all of the last four sections (relocation, line number, symbol table, and string table) may be missing from the final executable file. If the program is linked with the **-s** flag, all four of these sections will be absent. The line number information does not appear unless the program is compiled with the **-q** or **-q1** debug flag or assembled with the **-L** line number flag. If there are no unresolved external symbols after linking, the relocation information is no longer needed and is thus absent. If there are no symbols with names longer than eight characters, the string table is not needed and is thus absent.

1.4.1. Definitions and Conventions

Before continuing, you should become familiar with the following terms and conventions.

1.4.1.1. Sections

A section is the smallest portion of an object file that is relocated and able to be positioned at an independent location in memory. In the default case, there are three named sections with the names **.text**, **.data**, and **.bss**. In a Sierra Systems executable file, there is also usually a fourth section with the name **.ld_tbl**. TI-89 / TI-92 Plus apps and ASMs also have a **.const** section, and the **.ld_tbl** section is unused. Additional named and unnamed sections, up to a total of 126, can also be added.

1.4.1.2. Physical and Virtual Addresses

The address of a section or symbol is the offset of that section or symbol from address zero of the address space. The *physical address* is the actual location in memory where a section is loaded. The *virtual address* of a section is the address from which the section's data will execute. When a section's data has to be copied from ROM to RAM at program startup, the ROM copy (source) of the section's data is considered to be at the physical address and the RAM copy (destination) of the section's data is considered to be at the virtual address. The section header contains two address fields: a physical address and a virtual address. In the case where a section's data gets copied from ROM to RAM, the physical and virtual address entries will be different; but in the case where a program executes out of ROM, the physical and virtual address entries for the section's data will be the same.

1.4.1.3. C Language COFF File Structures

The C language COFF file structures and macro definitions used internally by the various Sierra Systems utilities that operate on object files are defined in the file **file_fmt.txt** supplied with the TI-89 / TI-92 Plus SDK. Note that structure members of type (**unsigned**) **char**, (**unsigned**) **short**, and (**unsigned**) **long** are defined as **unsigned char** arrays of length 1, 2, and 4, respectively. The structure members are represented as arrays of **unsigned char** to facilitate generating identical object files on host machines with different internal byte orderings. Machine-dependent macros for reading and writing the structures in **file_fmt.txt** are defined in file **com_fmt.txt**, also supplied with the TI-89 / TI-92 Plus SDK.

1.4.2. File Header

The file header contains the 20 bytes of information shown in Table 1.3. The last two bytes are flags that are used by the linker and other object file utilities. Defined in **file_fmt.txt** are **FILE_HDR** and **FILE_HDR_SIZE**, the typedefs for the file header structure and file header size, respectively.

Bytes	Declaration	Name	Description
0-1	**unsigned short**	**fh_magic**	Magic number, 0x150
2-3	**unsigned short**	**fh_nbr_sections**	Number of section headers (equals the number of sections)
4-7	**unsigned long**	**fh_time_date**	Time and date stamp indicating when the file was created (expressed as the number of seconds since 00:00:00 GMT, January 1, 1970)
8-11	**unsigned long**	**fh_symtab_ptr**	File pointer containing the starting offset of the symbol table
12-15	**unsigned long**	**fh_nbr_symtab_ents**	Number of entries in the symbol table
16-17	**unsigned short**	**fh_size_opt_hdr**	Number of bytes in the optional header
18-19	**unsigned short**	**fh_flags**	Flags (see Table 1.4)

Table 1.3: File Header Contents

1.4.2.1. Magic Number

The *magic number* specifies the target machine on which the object file can be executed. The number 0x150 identifies the object file as a 68000 family executable file. The number 0x150 is the only number recognized by the various Sierra Systems utilities; an error will be reported if a file that does not begin with 0x150 is encountered.

1.4.2.2. Optional Header Size

The optional header shown in Table 1.5 is system-dependent and not specified by the COFF standard. The size of the optional header is specified in the file header to allow system-independent object file utilities to skip past the optional header.

1.4.2.3. Flags

The last two bytes of the file header are flags that describe the type of the object file. The file header flags are described in Table 1.4.

Mnemonic	Flag	Meaning
FH_REL_STRPD	0x01	Relocation information stripped from file
FH_EXECUTABLE	0x02	File is executable (i.e., no unresolved references)
FH_LNNO_STRPD	0x04	Line numbers stripped from file
FH_LSYMS_STRPD	0x08	Local symbols stripped from file
FH_GSYMS_STRPD	0x10	Global symbols stripped from file
FH_ERR_IN_FILE	0x80	Error in object file

Table 1.4: File Header Flags

1.4.3. Optional Header

The optional header contains information that varies among the systems that use COFF. Applications place all system-dependent information in the optional header. As previously stated, system-independent COFF utilities can skip past the system-dependent optional header to perform tasks such as dumping the symbol table.

Bytes	Declaration	Name	Description
0-1	**unsigned short**	**magic**	Magic number, 0x107
2-3	**unsigned short**	**version**	Version stamp
4-7	**unsigned long**	**tsize**	Size of text in bytes
8-11	**unsigned long**	**dsize**	Size of initialized data in bytes
12-15	**unsigned long**	**bsize**	Size of uninitialized data in bytes
16-19	**unsigned long**	**entry**	Program entry point
20-23	**unsigned long**	**text_start**	Base address of text
24-27	**unsigned long**	**data_start**	Base address of data

Table 1.5: Sierra Systems Optional Header Contents

The Sierra Systems optional header is 28 bytes long, and the fields of the optional header are described in Table 1.5. The optional header is present only on linked executable files; it is not present on assembler-generated files or partially linked files (**-r** flag). Defined in **file_fmt.txt** are **A_OUT_HDR** and **A_OUT_HDR_SIZE**, the typedefs for the optional header structure and optional header size, respectively.

1.4.4. Section Headers

Every object file has one section header for each section in the file. The Sierra Systems object file has at least three section headers. The section headers describe the organization of data within the file. Table 1.6 describes the fields of the section header. Defined in **file_fmt.txt** are **SECTION_HDR** and **SECTION_HDR_SIZE**, the typedefs for the section header structure and section header size, respectively.

The size of a section is always padded to a multiple of four bytes.

The file pointers are byte offsets from the beginning of the file, and can be used to locate the start of the data, relocation, or line entries for the section.

Bytes	Declaration	Name	Description
0-7	**char**	**sh_name**	8-character null-padded section name
8-11	**unsigned long**	**sh_phys_addr**	Physical address of section
12-15	**unsigned long**	**sh_virt_addr**	Virtual address of section
16-19	**unsigned long**	**sh_size**	Section size in bytes
20-23	**unsigned long**	**sh_data_ptr**	File pointer to raw data
24-27	**unsigned long**	**sh_reloc_ptr**	File pointer to relocation entries
28-31	**unsigned long**	**sh_line_nbr_ptr**	File pointer to line number entries
32-33	**unsigned short**	**sh_nbr_reloc_ents**	Number of relocation entries
34-35	**unsigned short**	**sh_nbr_line_ents**	Number of line number entries
36-39	**unsigned long**	**sh_flags**	Flags (see Table 1.7)

Table 1.6: Section Header Contents

The *flags* field defines the type of the section. Table 1.7 lists the definitions of the section header flags.

Mnemonic	Flag	Meaning
SH_REG	0x000	Regular section: alloc'd, not reloc'd, load'd
SH_DSECT	0x001	Dummy section: not alloc'd, reloc'd, not load'd
SH_NOLOAD	0x002	No load section: alloc'd, reloc'd, not load'd
SH_RESIDENT	0x002	Resident section: alloc'd, reloc'd, not load'd
SH_GROUP	0x004	Grouped section: formed from input sections
SH_PAD	0x008	Padding section: not alloc'd, not reloc'd, load'd
SH_FILLONLY	0x008	Fill only section: filled at run-time
SH_COPY	0x010	Copy section: copied at run-time from destination address to virtual address
SH_TEXT	0x020	Section contains executable text
SH_DATA	0x040	Section contains initialized data
SH_BSS	0x080	Section contains only uninitialized data
SH_ORG	0x100	Section contains **ORG**'d (absolute) data
SH_INFO	0x200	Comment section: not alloc'd, not reloc'd, not load'd
SH_OVERT	0x400	Overlay section: not alloc'd, reloc'd, not load'd
SH_LIB	0x800	Library section

Table 1.7: Section Header Flags

All sections, including sections with uninitialized data such as section **.bss** (blank static storage), have an associated section header. Uninitialized sections have symbols that refer to them and symbols that are defined in them, but they do not have any relocation entries, line number entries, or data associated with them. Therefore, uninitialized sections have a section header, but occupy no other space in the object file. In the case of an uninitialized section, zeroes appear in the fields for the number of relocation and line number entries, as well as in the fields for all the file pointers.

1.4.5. Relocation Information

Object files have one relocation entry for each relocatable reference in a text-type or data-type section. The relocation entries are automatically generated by the assembler, and the information is used by the linker to resolve external references. Table 1.8 describes the information carried in the object file for each relocatable reference. Defined in **file_fmt.txt** are **RELOC_INFO** and **RELOC_INFO_SIZE**, the typedefs for the relocation entry structure and relocation entry size, respectively.

Bytes	Declaration	Name	Description
0-3	**unsigned long**	**r_virt_addr**	Virtual address of reference
4-7	**unsigned long**	**r_sym_index**	Symbol table index
8-9	**unsigned short**	**r_type_info**	Relocation type (see Table 1.9)

Table 1.8: Relocation Section Contents

The first field of the relocation entry is the virtual address of the text or data to which the entry applies. The 1-, 2-, or 4-byte area referenced by this first field is known as a hole since the assembler cannot resolve the reference. The address or PC-relative offset is ultimately determined and written to the hole by the linker. The second field is the index of the symbol table entry for the symbol that is being referenced. The third field indicates the type of relocation that is to be applied. Table 1.9 lists the relocation types supported by Sierra Systems.

Mnemonic	Flag	Meaning
RL_FIXED	0x00	Reference is absolute, no relocation is necessary, the entry will be ignored
RL_DIR_BYTE	0x0F	Direct 8-bit reference to symbol's virtual address
RL_DIR_WORD	0x10	Direct 16-bit reference to symbol's virtual address
RL_DIR_LONG	0x11	Direct 32-bit reference to symbol's virtual address
RL_PC_BYTE	0x12	A PC-relative 8-bit reference to symbol's virtual address
RL_PC_WORD	0x13	A PC-relative 16-bit reference to symbol's virtual address
RL_PC_LONG	0x14	A PC-relative 32-bit reference to symbol's virtual address
RL_CMPLX_OP	0x40	Operator in a complex relocatable expression
RL_CMPLX_ABS	0x50	Absolute operand in a complex relocatable expression
RL_CMPLX_REL	0x60	Relocatable operand in a complex relocatable expression
RL_CMPLX_EXT	0x70	External operand in a complex relocatable expression

Table 1.9: Relocation Types

1.4.5.1. Relocation

There are two types of relocation: absolute and PC-relative. The assembler reduces an unresolved reference to an offset from a relocatable section or an offset from an external symbol. The offset is saved in the hole and the index of the symbol (external symbol or section name symbol) is saved in the relocation entry. In addition, if the reference is PC-relative, the address or section-relative offset of the hole is subtracted from the contents of the hole.

The linker, which knows the addresses of all symbols, adds the address of the symbol referenced by the **r_sym_index** field to the contents of the hole. In addition, if the reference is PC-relative and the hole is in a relocatable section, the base address of this section is subtracted from the contents of the hole.

In the case of byte or word references (1-byte or 2-byte holes, respectively), it is possible that the final value determined by the linker will fit in the hole but the intermediate value to be filled in by the assembler will not fit. Sierra Systems has added an extension to the COFF standard to remove this deficiency. In the relocation entry, the higher order byte of the **r_type_info** field and the highest order byte of the **r_sym_index** field (assuming fewer than 16 million symbols) are unused. In the case of a 1-byte or 2-byte hole, the overflow first goes into the unused byte in **r_sym_index** and then into the unused byte in **r_type_info** effectively providing the assembler with a 24-bit or 32-bit hole, respectively.

1.4.5.2. Complex Relocation

A complex relocatable expression is an expression that cannot be reduced at assembly time to either an absolute value or a section-relative reference. For example, an expression that references multiple external symbols and/or symbols from different relocatable sections would be classified complex relocatable.

A complex relocatable expression is saved in a sequence of auxiliary relocation entries that follow the primary entry for the relocation hole. A separate entry is used for each operand and each operator in the expression. Table 1.10 describes the different types of complex relocation entries. The complex expression contained in the auxiliary relocation entries is subsequently evaluated by the linker, and the result is placed in the hole referenced by the primary entry.

	COFF Relocation Entry		
	r_virt_addr	**r_sym_index**	**r_type_info**
Primary Entry	hole address	entry count	type and size
Operator	operator code	–	**RL_CMPLX_OP**
Absolute Operand	value	–	**RL_CMPLX_ABS**
Reloc Operand	section offset	section index	**RL_CMPLX_REL**
External Operand	–	symbol index	**RL_CMPLX_EXT**

Table 1.10: Complex Relocation Entries

Note that the field **r_type_info** for the primary entry has the **RL_CMPLX_OP** bit set to designate the start of a complex relocatable expression.

1.4.6. Line Number Information

The line number information in the object file is used for source-level debugging. Line number information is present when C files are compiled with the **-q** or **-q1** debug flag or assembly language files are assembled with the **-L** line number flag. When compiled or assembled with one of the above listed flags, a line number entry is generated for every line of C or assembly language code. The line number entries are grouped by section and within each section grouping the entries are grouped by function as shown in Table 1.11.

symbol index	0
virtual address	line number
virtual address	line number
. . .	. . .
symbol index	0
virtual address	line number
virtual address	line number

Table 1.11: Line Number Grouping

Table 1.12 describes the fields within a line number entry. Defined in **file_fmt.txt** are **LINE_NBR** and **LINE_NBR_SIZE**, the typedefs for the line number entry structure and line number entry size, respectively.

Bytes	Declaration	Name	Description
0-3	**unsigned long**	**l_sym_index**	Symbol table index of function
0-3	**unsigned long**	**l_phys_addr**	Address of source line
4-5	**unsigned short**	**l_line_nbr**	Source line number

Table 1.12: Line Number Section Contents

The first line number entry within a function grouping specifies line number 0 and has, in place of the virtual address, an index into the symbol table for the entry containing the name of the function. The subsequent entries have the actual line numbers relative to the open brace ' { ' that begins the function, and the address of the text that corresponds to the line number. The line number entries are ordered by increasing address.

The **.bf** (begin function) symbol entry immediately follows the auxiliary entry for the function name symbol. The absolute C source line number of the function's open brace is specified in the auxiliary entry for the **.bf** symbol (see section **1.4.9.4 Beginning of Blocks and Functions**). Absolute C source line numbers

for all other lines in a function are computed by offsetting the relative line numbers in the line number entries by the absolute line number in the **.bf** symbol's auxiliary entry.

1.4.7. Symbol Table

Because of symbolic debugging requirements, the order of the symbols within the symbol table is very important. The symbols appear in the order shown in Table 1.13.

filename 1
function 1
local symbols for function 1
function 2
local symbols for function 2
. . .
statics for file 1
. . .
filename 2
local symbols for function 1
. . .
statics for file 2
. . .
all defined global symbols
all undefined global symbols

Table 1.13: COFF Symbol Table

Local symbols for a function are the symbols that are defined within a function and accessible only within that function. The term statics as used in Table 1.13 identifies symbols that are C language variables of storage class **static** defined outside any function. The symbol table consists of at least one 18-byte entry per symbol with some symbols followed by one or more auxiliary entries also 18 bytes each. A symbol table entry contains the name of the symbol (or file offset to the name), the value, the type, and other information.

1.4.7.1. Special Symbols

Included in the symbol table are special symbols that are generated by the compiler and assembler. Most of the special symbols are needed for source-level debugging. Table 1.14 lists the special symbols.

Symbol	Meaning
.file	Filename
.text	Address of **.text** section
.data	Address of **.data** section
.bss	Address of **.bss** section
.bb	Address of start of inner block
.eb	Address of end of inner block
.bf	Address of start of function
.ef	Address of end of function
.**target**	Pointer to structure or union returned by function
.*x*fake	Dummy tag name for structure, union, or enumeration
.eos	End of members of structure, union, or enumeration
_etext	Next available address after the end of the output section **.text**
_edata	Next available address after the end of the output section **.data**
_end	Next available address after the end of the output section **.bss**

Table 1.14: Special Symbols in the Symbol Table

Six of the special symbols are used in pairs. The **.bb** and **.eb** symbols encapsulate the symbols defined in inner blocks. The **.bf** and **.ef** symbols encapsulate each function. The **.*x*fake** and **.eos** symbols define the limits of unnamed structures, unions and enumerations. The **.eos** symbol is also paired with actual names to define the limits of named structures, unions, and enumerations.

When a structure, union, or enumeration is defined without a tag, the compiler automatically generates a name for internal use. The generated name is **.*x*fake**, where ***x*** is a unique decimal number. In the case where a file defined three unnamed structures, structure tags with the names **.0fake**, **.1fake**, and **.2fake** would be generated.

1.4.7.2. Inner Blocks

The C language defines a block as a compound statement that begins with an open brace ' { ' and ends with a balancing close brace ' } '. An inner block is a block that exists within a function (which is also a block). For each inner block that has local symbols defined in it, the special symbol **.bb** is inserted in the symbol table immediately before the first local symbol of the block. Analogously, the special symbol **.eb** is inserted in the symbol table immediately after the last local symbol of the block. Because inner blocks can be nested to multiple levels, the **.bb** – **.eb** symbol pairs and associated symbols can also be nested.
Table 1.15 shows an example of nested C language blocks and the associated symbol table.

Nested Blocks	Symbol Table
func1(int a)	_func1
{	.bf
int b;	a
{	b
int c;	.bb
{	c
int d;	.bb
}	d
}	.eb
}	.eb
	.ef
func2(int e)	_func2
{	.bf
int f;	e
int g;	f
{	g
int h;	.bb
}	h
}	.eb
	.ef

Table 1.15: Example Symbol Table for Functions and Nested Blocks

1.4.7.3. Symbols and Functions

For each function, a special symbol **.bf** is put between the function name and the first local symbol of the function name in the symbol table. Correspondingly, a special symbol **.ef** is put immediately after the last symbol of the function in the symbol table. Associated with the **.bf** and **.ef** symbols (as defined by their auxiliary symbol table entries) are the absolute line numbers of the function's open brace ' { ' and close brace ' } ', respectively. The example in Table 1.15 (in addition to showing nested inner blocks) shows a pair of C language functions and the associated symbol table.

1.4.8. Symbol Table Entries

All symbols, regardless of their type and storage class, use the same symbol table format. Every symbol table entry occupies 18 bytes. Table 1.16 describes the fields within a symbol table entry. It should be noted that the indices for symbol table entries begin with 0 (not 1). Also, auxiliary entries count as symbol entries for purposes of indexing into the symbol table. Defined in **file_fmt.txt** are **SYM_ENT** and **SYM_ENT_SIZE**, the typedefs for the symbol table entry structure and symbol table entry size, respectively.

Bytes	Declaration	Name	Description
0-7	**char**	**sym_name**	8-character null padded symbol name
0-3	**unsigned long**	**sym_zeroes**	Zero in this field indicates the name is in the string table
4-7	**unsigned long**	**sym_offset**	Offset of the name in the string table
8-11	**unsigned long**	**sym_value**	Symbol value (storage class dependent)
12-13	**unsigned short**	**sym_sec_nbr**	Section number of symbol
14-15	**unsigned short**	**sym_type**	Basic and derived type information
16	**char**	**sym_sclass**	Storage class of symbol
17	**char**	**sym_nbr_aux**	Number of auxiliary entries

Table 1.16: Symbol Table Entry

1.4.8.1. Symbol Names

The symbol name resides either within the 18 byte symbol table entry itself or in a string table at the end of the object file. The first eight bytes of the symbol table entry are a union of an eight byte character array and two longs. If the symbol name is eight or fewer characters in length, the symbol name (null padded) is stored in the first eight bytes of the symbol table entry. If the symbol name is longer than eight characters, the entire null-terminated symbol name is stored in the string table. When the symbol name is stored in the string table, the first four bytes of the symbol table entry are zero and the second four bytes contain the offset (relative to the beginning of the string table) of the name in the string table. Since there cannot be a symbol with a null name, the zeroes in the first four bytes distinguish a symbol table entry with an offset into the string table from a symbol table entry with a name in the first eight bytes.

1.4.8.2. Storage Class

The storage class field is assigned one of the values described in Table 1.17. The mnemonics listed in the table are defined in **file_fmt.txt**.

All the storage classes that appear in the object file except for **C_ALIAS** are generated by the assemblers, **asm68** and **asm68k**. The storage class **C_ALIAS** is generated by the linker, **link68** (**-P** flag not specified), when it removes duplicate structure, union and enumeration definitions and places alias entries in their places.

Note that not all the storage classes listed in Table 1.17 appear in the object file. Some of the storage classes such as **C_EFCN**, **C_ARRAY**, **C_SUE**, and **C_SKIP** are used only internally by the compiler and assemblers.

C_EFCN	-1	Physical end of function
C_NULL	0	
C_AUTO	1	Automatic variable
C_EXT	2	External symbol
C_STAT	3	Static
C_REG	4	Register variable
C_EXTDEF	5	External definition
C_LABEL	6	Label
C_ULABEL	7	Undefined label
C_MOS	8	Member of structure
C_ARG	9	Function argument
C_STRTAG	10	Structure tag
C_MOU	11	Member of union
C_UNTAG	12	Union tag
C_TPDEF	13	Type definition
C_USTATIC	14	Uninitialized static
C_ENTAG	15	Enumeration tag
C_MOE	16	Member of enumeration
C_REGPARM	17	Register parameter
C_FIELD	18	Bit field
C_ARRAY	19	Array dimension information
C_SUE	20	Structure, union, or enumeration
C_SKIP	21	Symbol that should not be output
C_BLOCK	100	Beginning and end of block
C_FCN	101	Beginning and end of function
C_EOS	102	End of structure
C_FILE	103	Filename
C_ALIAS	105	Duplicate tag
C_HIDDEN	106	Like static, used to avoid name conflicts

Table 1.17: Storage Classes

1.4.8.3. Storage Classes for Special Symbols

Some of the special symbols listed in Table 1.14 are restricted to certain storage classes. Table 1.18 lists the restricted special symbols with their allowed storage classes. Also, the storage classes **C_FILE**, **C_BLOCK**, **C_FCN**, and **C_EOS** are used only with the special symbols they are shown associated with in Table 1.18.

Special Symbol	Storage Class
.file	**C_FILE**
.bb	**C_BLOCK**
.eb	**C_BLOCK**
.bf	**C_FCN**
.ef	**C_FCN**
.target	**C_AUTO**
.*x*fake	**C_STRTAG, C_UNTAG, C_ENTAG**
.eos	**C_EOS**
.text	**C_STAT**
.data	**C_STAT**
.bss	**C_STAT**

Table 1.18: Storage Class of Special Symbols

1.4.8.4. Symbol Value Field

The interpretation of the value of a symbol is a function of the symbol's storage class. Table 1.19 summarizes the relationship between storage class and value.

Storage Class	Meaning of Value
C_AUTO	Stack offset in bytes
C_EXT	Relocatable address
C_STAT	Relocatable address
C_REG	Register number
C_LABEL	Relocatable address
C_MOS	Offset in bytes
C_ARG	Stack offset in bytes
C_STRTAG	0
C_MOU	0
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_EOS	Size
C_FILE	(see text below)
C_ALIAS	Tag index
C_HIDDEN	Relocatable address

Table 1.19: Storage Class and Value

If a symbol has storage class **C_FILE**, the value of the symbol is the symbol table entry index of the next **.file** symbol. The **.file** entries form a one-way linked list within the symbol table. The value of the last **.file** symbol table entry is the index of the first global symbol.

Relocatable symbols have a value that is equal to the virtual address of the symbol. When a section is relocated by the linker, the values of the section's relocatable symbols change.

1.4.8.5. Section Number Field

Section numbers are listed in Table 1.20.

Mnemonic	Section Number	Meaning
N_DEBUG	-2	Special symbolic debugging symbol
N_ABS	-1	Absolute value
N_UNDEF	0	Undefined external symbol
N_SCNUM	1-126	Section number where symbol is defined

Table 1.20: Section Number

Section number -2 identifies symbolic debugging symbols, including structure, union and enumeration tag names, typedefs, and filenames. Section number -1 identifies symbols that have a non-relocatable (absolute) value. Examples of absolute-valued symbols include automatic and register variables, function arguments, structure members and **.eos** symbols. The **.text**, **.data**, and **.bss** sections typically default to section numbers 1, 2, and 3, respectively.

With one exception, section number 0 identifies a relocatable external symbol that is not defined in the current file. The one exception is the external symbol generated as the result of defining an uninitialized external variable. The ANSI C standard permits only a single defining instance (initialized or uninitialized) of a particular variable. To permit compatibility with early C environments, multiple defining instances of uninitialized variables are permitted when the linker is invoked with the **-C** command line flag. In each file where the symbol for the uninitialized variable is defined, the section number of the symbol is 0, and the value of the symbol is representative of the size and alignment of the symbol. The size of the symbol occupies bits 0 – 29 and the symbol alignment occupies bits 30 and 31. The alignment is a Sierra Systems extension to COFF where bits 30 and 31 low indicate quad alignment, bit 30 high (31 low) indicates even alignment, and bit 31 high (30 low) indicates no alignment required. When the files are combined into an executable object file, the linker (**-C** flag specified) combines all the symbols of the same name into one symbol in the **.bss** section. The maximum size of all the input symbols with the same name is used to allocate space for the symbol and the value becomes the address of the symbol. This is the only case where a symbol has a section number 0 and a non-zero value.

1.4.8.6. Section Numbers and Storage Classes

Symbols having certain storage classes are also restricted to having certain section numbers. Table 1.21 summarizes the relationship between storage class and section number.

Storage Class	Section Number
C_AUTO	N_ABS
C_EXT	N_ABS, N_UNDEF, N_SCNUM
C_STAT	N_SCNUM
C_REG	N_ABS
C_LABEL	N_UNDEF, N_SCNUM
C_MOS	N_ABS
C_ARG	N_ABS
C_STRTAG	N_DEBUG
C_MOU	N_ABS
C_UNTAG	N_DEBUG
C_TPDEF	N_DEBUG
C_ENTAG	N_DEBUG
C_MOE	N_ABS
C_REGPARM	N_ABS
C_FIELD	N_ABS
C_BLOCK	N_SCNUM
C_FCN	N_SCNUM
C_EOS	N_ABS
C_FILE	N_DEBUG
C_ALIAS	N_DEBUG

Table 1.21: Section Number and Storage Class

1.4.8.7. Type Entry

The type field in the symbol table entry contains information on the basic and derived type for the symbol. The type information is generated for a symbol only if the files are generated with symbolic debugging in mind. Type information is generated when the compiler is invoked with the **-q** or **-q1** flag and/or the assembler is invoked with the **-L** flag and **.type** directives are inserted into the assembly language file. Each symbol has one basic or fundamental type and from zero to six derived types. The following is the format of the 16-bit type entry:

d6	d5	d4	d3	d2	d1	type

Bits 0 through 3, identified as **type**, indicate one of the fundamental types listed in Table 1.22. Bits 4 through 15 are arranged as six 2-bit fields identified as **d1** through **d6**. The fields **d1** through **d6** represent levels of the derived types listed in Table 1.23.

Mnemonic	Value	Type
T_NULL	0	Type not assigned
T_LDOUBLE	1	Long double
T_CHAR	2	Character
T_SHORT	3	Short
T_INT	4	Integer
T_LONG	5	Long integer
T_FLOAT	6	Float
T_DOUBLE	7	Double
T_STRUCT	8	Structure
T_UNION	9	Union
T_ENUM	10	Enumeration
T_MOE	11	Member of enumeration
T_UCHAT	12	Unsigned character
T_USHORT	13	Unsigned short
T_UINT	14	Unsigned integer
T_ULONG	15	Unsigned long integer

Table 1.22: Fundamental Types

Mnemonic	Value	Type
DT_NON	0	No derived type
DT_PTR	1	Pointer
DT_FCN	2	Function
DT_ARRAY	3	Array

Table 1.23: Derived Types

The following two examples demonstrate the interpretation of the symbol table entry type field.

```
int **func1()
```

In the above example, **func1** is a function that returns a pointer to a pointer to an integer. The fundamental type of **func1** is 4 (integer), the first derived type is 2 (function), the second derived type is 1 (pointer) and the third derived type is also 1 (pointer). Combining the information into a single word (01 01 10 0100), as previously described, the hexadecimal representation of the value in the type field is 0x164.

```
short (*(*pfunc[2][3])())[4]
```

In this example, **pfunc** is a two-dimensional array of pointers to a function that returns a pointer to an array of short integers. The fundamental type of **pfunc** is 3 (short integer), the first derived type is 3 (array), the second derived type is 3 (array), the third derived type is 1 (pointer, the fourth derived type is 2 (function), the fifth derived type is 1 (pointer) and the sixth derived type is again 3 (array). Combining the information into a single word (11 01 10 01 11 11 0011), as previously described, the hexadecimal representation of the value in the type field is 0xD9F3.

1.4.8.8. Type Entries and Storage Classes

Symbols having certain storage classes are also restricted to having certain type entries. Table 1.24 summarizes the relationship between storage class and type entries.

Storage Class	Derived Type			Fundamental Type
	Function	Array	Pointer	
C_AUTO	no	yes	yes	Any except **T_MOE**
C_EXT	yes	yes	yes	Any except **T_MOE**
C_STAT	yes	yes	yes	Any except **T_MOE**
C_REG	no	no	yes	Any except **T_MOE**
C_LABEL	no	no	no	**T_NULL**
C_MOS	no	yes	yes	Any except **T_MOE**
C_ARG	yes	no	yes	Any except **T_MOE**
C_STRTAG	no	no	no	**T_STRUCT**
C_MOU	no	yes	yes	Any except **T_MOE**
C_UNTAG	no	no	no	**T_UNION**
C_TPDEF	no	yes	yes	Any except **T_MOE**
C_ENTAG	no	no	no	**T_ENUM**
C_MOE	no	no	no	**T_MOE**
C_REGPARM	no	no	yes	Any except **T_MOE**
C_FIELD	no	no	no	**T_ENUM**, **T_CHAR**, **T_UCHAR**, **T_SHORT**, **T_USHORT**, **T_INT**, **T_UINT**, **T_LONG**, **T_ULONG**
C_BLOCK	no	no	no	**T_NULL**
C_FCN	no	no	no	**T_NULL**
C_EOS	no	no	no	**T_NULL**
C_FILE	no	no	no	**T_NULL**
C_ALIAS	no	no	no	**T_STRUCT**, **T_UNION**, **T_ENUM**

Table 1.24: Type Entries by Storage Class

Conditions for the derived types shown in Table 1.24 apply to all derived types, **d1** through **d6**, with the added restriction that it is impossible to have two consecutive derived types of function.

1.4.9. Auxiliary Table Entries

An auxiliary entry of a symbol contains the same number of bytes (18) as the symbol table entry. However, unlike symbol table entries, the format of an auxiliary entry is a function of the symbol's type and storage class. Table 1.25 describes the relationship between type and storage class, and the formation of the auxiliary entry.

Symbol Name	Storage Class	Type Entry		Auxiliary Entry Format
		d1	type	
.file	**C_FILE**	**DT_NON**	**T_NULL**	Filename
section name	**C_STAT**	**DT_NON**	**T_NULL**	Section
tag name	**C_STRTAG**, **C_UNTAG**, **C_ENTAG**	**DT_NON**	**T_NULL**	Tag name
.eos	**C_EOS**	**DT_NON**	**T_NULL**	End of structure
function name	**C_EXT**	**DT_FCN**	(note 1)	Function
array name	(note 2)	**DT_ARY**	(note 1)	Array
.bb	**C_BLOCK**	**DT_NON**	**T_NULL**	Beginning of blocks and functions
.bf	**C_FCN**	**DT_NON**	**T_NULL**	Beginning of blocks and functions
.eb	**C_BLOCK**	**DT_NON**	**T_NULL**	End of blocks and functions
.ef	**C_FCN**	**DT_NON**	**T_NULL**	End of blocks and functions
name related to structure, union, or enumeration	(note 2)	**DT_PTR, DT_ARR**	**T_STRUCT T_UNION**	Structure, union, enumeration

Note 1: Any except **T_MOE**
Note 2: **C_AUTO**, **C_STAT**, **C_MOS**, **C_MOU**, **C_TPDEF**

Table 1.25: Auxiliary Symbol Table Entries

In Table 1.25, *tag name* indicates any symbol name including ***.xfake***. The symbol names *function name* and *array name* represent any name for a function or array respectively. Any symbol that satisfies more than one condition listed in Table 1.25 (e.g., array of structures) should have a union format in the auxiliary entry. Any symbol that does not satisfy any of the conditions listed in Table 1.25 should not have one of the auxiliary entries listed in Tables 1.26 through 1.34. Future extensions may allow symbols to have more than one auxiliary entry and auxiliary entries that are not listed in the above referenced tables.

Defined in **file_fmt.txt** is **AUX_ENT**, the typedef for the auxiliary symbol table entry structure.

1.4.9.1. Filenames

The auxiliary symbol table entries for filenames have the format shown in Table 1.26.

Bytes	Declaration	Name	Description
0-13	**char**	**aux_file_name**	14-character null padded filename
14-17	–	–	Unused (filled with 0's)

Table 1.26: Auxiliary Symbol Entry for Filenames

1.4.9.2. Sections

The auxiliary symbol table entries for sections have the format shown in Table 1.27.

Bytes	Declaration	Name	Description
0-3	**unsigned long**	**sec.len**	Section length
4-5	**unsigned short**	**sec.nbr_rel_ents**	Number of relocation entries
6-7	**unsigned short**	**sec.nbr_line_nbrs**	Number of line number entries
8-17	–	–	Unused (filled with 0's)

Table 1.27: Auxiliary Symbol Entry for Sections

1.4.9.3. Functions

The auxiliary symbol table entries for functions have the format shown in Table 1.28. The *parameter-and-fp-format* word in the function auxiliary entry contains information on how function parameters are pushed onto the stack and which floating-point format the compiler is generating code for. The bits in the *parameter-and-fp-format* word are interpreted as follows when set:

Bit 0 – Function is prototyped
Bit 1 – Parameters of type **short** and **char** pushed as four-bytes when prototyped (Parameters of type **short** and **char** are always pushed as two-bytes for TI calculators.)
Bit 2 – Parameters of type **float** pushed as 10 bytes (**double**) when prototyped
Bit 3 – Not supported by Texas Instruments.
Bit 4 – Not supported by Texas Instruments.
Bit 5 – Not supported by Texas Instruments.

Bytes	Declaration	Name	Description
0-3	**unsigned long**	**symbol.tag_index**	Tag index
4-7	**unsigned long**	**symbol.u1.func_size**	Size of function
8-11	**unsigned long**	**symbol.u2.s.line_ptr**	File pointer to line number
12-15	**unsigned long**	**symbol.u2.s.end_index**	Index of entry after this point
16-17	**unsigned short**	**symbol.high_size**	Parameter and fp format info

Table 1.28: Auxiliary Symbol Entry for Functions

1.4.9.4. Beginning of Blocks and Functions

The auxiliary symbol table entries for beginning of blocks and functions have the format shown in Table 1.29.

Bytes	Declaration	Name	Description
0-3	–	–	Unused (filled with 0's)
4-5	**unsigned short**	**symbol.u1.s.c_line_nbr**	C-source line number
6-11	–	–	Unused (filled with 0's)
12-15	**unsigned long**	**symbol.u2.s.end_index**	Index of entry after this block
16-17	–	–	Unused (filled with 0's)

Table 1.29: Auxiliary Symbol Entry for Beginning of Blocks and Functions

1.4.9.5. End of Blocks and Functions

The auxiliary symbol table entries for the end of blocks and functions have the format shown in Table 1.30.

Bytes	Declaration	Name	Description
0-3	–	–	Unused (filled with 0's)
4-5	**unsigned short**	**symbol.u1.s.c_line_nbr**	C-source line number
6-17	–	–	Unused (filled with 0's)

Table 1.30: Auxiliary Symbol Entry for End of Blocks and Functions

1.4.9.6. Arrays

The auxiliary table entries for arrays have the format shown in Table 1.31.

Bytes	Declaration	Name	Description
0-3	**unsigned long**	**symbol.tag_index**	Tag index
4-5	**unsigned short**	**symbol.u1.s.c_line_nbr**	Line number of declaration
6-7	**unsigned short**	**symbol.u1.s.size**	Size of array
8-9	**unsigned short**	**symbol.u2.array_dim[0]**	First dimension
10-11	**unsigned short**	**symbol.u2.array_dim[1]**	Second dimension
12-13	**unsigned short**	**symbol.u2.array_dim[2]**	Third dimension
14-15	**unsigned short**	**symbol.u2.array_dim[3]**	Fourth dimension
16-17	–	–	Unused (filled with 0's)

Table 1.31: Auxiliary Symbol Entry for Arrays

1.4.9.7. Tag Names

The auxiliary symbol table entries for tag names have the format shown in Table 1.32.

Bytes	Declaration	Name	Description
0-5	–	–	Unused (filled with 0's)
6-7	**unsigned short**	**symbol.u1.s.size**	Size of structure, union, or enumeration
8-11	–	–	Unused (filled with 0's)
12-15	**unsigned long**	**symbol.u2.s. end_index**	Index of next entry beyond this structure, union, or enumeration
16-17	–	–	Unused (filled with 0's)

Table 1.32: Auxiliary Symbol Entry for Tag Names

1.4.9.8. End of Structures

The auxiliary symbol table entries for the end of structures have the format shown in Table 1.33.

Bytes	Declaration	Name	Description
0-3	**unsigned long**	**symbol.tag_index**	Tag index
4-5	–	–	Unused (filled with 0's)
6-7	**unsigned short**	**symbol.u1.s.size**	Size of structure, union, or enumeration
8-17	–	–	Unused (filled with 0's)

Table 1.33: Auxiliary Symbol Entry for End of Structures

1.4.9.9. Names Related to Structures, Unions, and Enumerations

The auxiliary table entries for structure, union, and enumeration symbols have the format shown in Table 1.34.

Bytes	Declaration	Name	Description
0-3	**unsigned long**	**symbol.tag_index**	Tag index
4-5	–	–	Unused (filled with 0's)
6-7	**unsigned short**	**symbol.u1.size**	Size of structure, union, or enumeration
8-17	–	–	Unused (filled with 0's)

Table 1.34: Auxiliary Symbol Entry for Structures, Unions, Enumerations

1.4.10. String Table

Symbol table names longer than eight characters are stored contiguously in the string table with each symbol name terminated with a null character. The string table is located immediately after the symbol table and it can be located by adding the size of the symbol table to the base offset of the symbol table. The first four bytes of the string table contain the size of the string table in bytes; thus the first string table symbol will be at offset four.

For example, given a file that includes two symbols with names longer than eight characters, *this_is_a_long_name* and *this_is_a_still_longer_name*, the string table will appear as shown in Table 1.35.

0	0	0	52	't'	'h'	'i'	's'	'_'	'i'
's'	'_'	'a'	'_'	'l'	'o'	'n'	'g'	'_'	'n'
'a'	'm'	'e'	'\0'	't'	'h'	'i'	's'	'_'	'i'
's'	'_'	'a'	'_'	's'	't'	'i'	'l'	'l'	'_'
'l'	'o'	'n'	'g'	'e'	'r'	'_'	'n'	'a'	'm'
'e'	'\0'								

Table 1.35: Example String Table

The string table offset of *this_is_a_long_name* is 4, and the string table offset of *this_is_a_still_longer_name* is 24.

Section 2: Compiler

Figures

Tables

2. Compiler

2.1. Introduction

The Sierra Systems C compiler, **com68**, converts C language statements into 68000 assembly code. It generates highly efficient code and supports ROMable and position-independent code generation for the 68000.

The C compiler, **com68**, was developed by Sierra Systems to support certain Motorola processors and coprocessors and IEEE format floating-point numbers. Under license from Sierra Systems, Texas Instruments has modified this software to support TI BCD floating-point numbers, and support for coprocessors has been removed. Although the software has not been modified to exclude support for processors other than the 68000, the 68000 is the only processor supported by Texas Instruments. The license from Texas Instruments to use these products is restricted to development of software that is targeted to execute only on TI calculators.

2.2. Invoking the Compiler

Typically, the TI **FLASH** Studio™ will handle all invocations of the compiler, using the correct command line flags required to produce TI-89 / TI-92 Plus apps or ASMs. The following discussion of command line format and flags is included for developers who may wish to use **com68** directly from the command line or create their own makefile.

The C compiler is ordinarily invoked with two filenames listed on the command line. The first filename specifies the C source file and the second specifies the assembly language output file. If no output file is specified, the compiler writes the assembly language output to **stdout** (usually the display screen). If no input file is specified either, the compiler reads C source input from **stdin** (usually the keyboard).

The following command compiles the file test.c and creates the 68000 assembly language file test.s:

```
com68 test.c test.s
```

2.3. Command Line Flags

Flags are used to change the behavior of the compiler. They are specified on the command line along with the input and output filenames. Flags appear on the command line as strings of one or more alphabetic characters prefixed with a hyphen (-). Some flags require arguments, and some accept optional

arguments. Other flags take no arguments. Flags may be grouped together following a single hyphen; however, any flag that accepts an optional argument must be the last flag in a group. The TI-89 / TI-92 Plus SDK includes example invocations of the compiler. It is highly recommended that you only use the flags as shown in those files.

The following command compiles the C source from the file demo_in.c and places the generated 68000 assembly code into demo_out.s:

```
com68 -l -O demo_in.c demo_out.s
```

The two flags listed on the command line cause the C source to be listed in the output file as comments, and full optimization to be applied to the program, respectively.

The command can also be specified as follows without changing the behavior of the compiler:

```
com68 demo_in.c -lO demo_out.s
```

These two command lines demonstrate both how flags can be grouped together using a single hyphen (-) and that they can appear anywhere on the command line.

When two or more contradictory flags are specified on the command line, the last flag entered determines the behavior. For example, if the **-Op0** flag (turn off post-code-generation optimizations) is followed by the **-Op1** flag (turn on post-code-generation optimizations), the **-Op0** flag is ignored. This feature may be useful in some cases, for example, you may wish to override a flag contained in a file included with the **-i** flag.

2.3.1. Usage

The following is a summary of the command line usage information for the compiler (see **-u** flag):

usage:
```
com68 [-fCcas#] [-CEPQTlq#su] [-Idir] [-Dname[=def]]
[-Uname] [-Xabcdef#ilpr#uw#AC#ILNRSWs#E#2]
[-ZabcdefilpqruwAILNRSsW2] [-O[a#c#f#i#l#m#p#r#s#t#x#zA]]
[-i cmd_file] [infile [outfile]]
```

2.3.2. Default Behavior

If no command line flags are specified, the compiler defaults to the following settings:

68000 default settings: `com68 -fc3 -Oa2c0f0i1l1m1p1r2s2x0`

The command line flags for the default settings are defined as follows:

-fc3 When specified by the **-l** flag, C source lines with comments and blank lines stripped out are indented by three tabs and placed into the assembly language output listing.

-Oa2 Both **short** and **int** data are aligned to even boundaries.

-Oc0 Do not coalesce function call stack cleanups.

-Of0 Set up a stack frame only when necessary.

-Oi1 Generate in-line code for the **strcpy** function.

-Ol1 Move only constants out of inner loops. Full loop invariant optimizations are not performed.

-Om1 Expand multiplication by a constant into a sequence of shifts and adds if it results in faster execution.

-Op1 Perform all post-code-generation optimizations.

-Or2 Treat all qualifying automatic variables as register candidates.

-Os2 In the presence of a function prototype, arguments of type **char** and **short int** are pushed onto the stack as two-byte objects.

-Ox0 Do not perform common subexpression optimizations.

2.3.3. Description of Flags

-C Leave comments in the preprocessed C source. The **-E** flag is implied.

-D*name*[=*string*]

Define the name *name*. The '=' and substitution string *string* are optional; the name *name* is defined to be 1 if the optional string is omitted. No whitespace is allowed on either side of the '='.

-E Send the preprocessed C source to the output file. Do not compile. See also **-P** flag.

-f Format the source listings and source line numbers that are optionally inserted as comments into the assembly language output. The **-l** flag must be specified to enable the selected format.

- **a** Place C source line numbers next to the corresponding assembly language statements.
- **c** Insert C source lines with comments and blank lines stripped into the assembly language listing.

C Insert C source lines including comments into the assembly language output.

s Place line numbers in front of each C source line in the assembly language listing.

Specify the number of tabs used to indent the C source intermixed with the assembly language listing. The default is three tabs.

-i *cmd_file* Include the contents of the file *cmd_file* into the command line input at the current position in the command line. The **-i** flag cannot be used inside an included file; however, it can be specified multiple times on the command line.

-I *path* Search the directory specified by *path* for **#include** files before looking in the default directory(ies). There is no limit to the number of times the **-I** flag can be specified.

-l Mix C source and line numbers into the assembly language listing as specified by the **-f** format option.

-LIC Display serial number and license information.

-M
-M1
-M2[t*nbr***]**
-M3[t*nbr***]**
-M4[t*nbr***]**
-M5

Not supported by Texas Instruments, however, **-M**, **-M1**, **-M2[**t*nbr***]**, **-M3[**t*nbr***]**, **-M4[**t*nbr***]**, and **-M5** are recognized as reserved flags by the compiler.

-O Turn on or off specified compiler optimizations. Specifying no flags after the **-O** flag is equivalent to specifying **-Ol2x1c1t1** (loop invariant, common subexpression, coalescing function call stack cleanups, and tail recursion elimination). All other optimizations are enabled by default (see section **2.3.2 Default Behavior**).

a# Specify alignment for integer data types. **a1** imposes no alignment on any integer data types. **a2** aligns both **short** and **int** data types to even boundaries (default for 68000). **a4** aligns **short** data to even boundaries and **int** data to quad boundaries.

A Ignore the possibility of aliases in the register scorecarding optimization. When the **-OA** flag is specified, slightly smaller, faster code is generated; however, there is a small possibility that the code will be incorrect if aliases exist.

c# **c1** causes multiple function call stack cleanups to be coalesced into a single cleanup. **c0** inhibits function call stack cleanups from being coalesced (default).

f# **f0** specifies that a stack frame (**link** and **unlk**) should not be setup if it is not essential (default). **f1** specifies that a stack frame should always be setup.

i# **i1** causes calls to the **strcpy** function to generate in-line code (default). **i0** causes calls to the **strcpy** function to generate a normal function call. **i2** is the same as **i1** with the added provision that all arrays are aligned to the same boundary as an **int**.

l# Move expressions that are not modified inside a loop to outside the loop. **l0** specifies that no loop invariant expressions are to be moved. **l1** specifies that only constants are to be moved outside inner-most loops (default). The **-Op1** flag (also default) must also be in effect for **l1** to have any effect. **l2** specifies that invariant expressions are to be moved outside the inner-most loops only. **l3** specifies that invariant expressions are to be moved through as many levels of loop nesting as possible. **l4** specifies the same action as **l3**, except that a warning is issued (correct code is still generated) if the compiler exceeds the number of loop invariants that it can handle. **l2** typically gives the best performance when this message is encountered.

m# **m0** inhibits multiplication by a constant from being expanded into a sequence of shifts and adds; it also inhibits the move-multiple instruction from being expanded into discrete moves. **m1** allows multiplication by a constant to be expanded, but does not allow the move-multiple instruction to be expanded (default).

p# **p0** specifies that no post-code-generation optimizations are to be performed. **p1** specifies that post-code-generation optimizations are to be performed (default).

r# Specifies how automatic variables are selected for placement into machine registers. **r0** specifies that no variables are to be placed into registers. **r1** specifies that only variables declared with the register storage-class specifier are to be candidates for placement into registers. **r2** specifies that both variables declared register and those not declared register will be register candidates (default). The nonregister variables are selected as register candidates based on how often they are used. They are selected after the declared register candidates until all have been assigned register candidate status or a total of 32 (including those declared register) are selected. **r3** specifies the same action as **r2**, except that register declarations are ignored. Variables that were declared register may still be assigned to registers. **r4** prevents floating-point automatic variables that have not been declared register from being placed into floating-point registers; otherwise, the behavior is the same as described by **r2**. For more information, refer to section **2.15.1 Register Allocation**.

s# The **-Os#** flag determines how to push a prototyped function argument onto the stack. A **short int** or **char** function prototype results in a two-byte push when **s2** (default) is specified and a four-byte push when **s4** is specified. In the presence of a **float** or a **double** prototype, the argument is pushed as a **double** (10 bytes) and will be pushed on as a **double** (10 bytes) when **s3** or **s5** is specified (**s2** or **s4** implied, respectively).

t# **t0** disables tail recursion elimination (default). **t1** enables tail recursion elimination.

x# **x0** specifies that duplicate common subexpressions are not to be removed (default). **x1** specifies that references to different copies of a common subexpression are to be replaced by references to a single subexpression with duplicate subexpressions removed.

z Optimize the generated code for size, possibly at the expense of speed. **z** disables the in-line string copy routine, the unrolling of structure copy loops, and the expansion of multiplication by constants (except where the expanded code is more compact). The stack cleanup coalesce option is enabled. Except for disabling the unrolling of structure copy loops, which can only be accomplished with this flag, the **-Oz** flag is equivalent to **-Oi0c1m0**.

-p*proc* Not supported by Texas Instruments, however, **-p***proc* is recognized as a reserved flag by the compiler.

-P Strip **#line** directives and multiple sequential blank lines out of preprocessed C source files. The **-E** flag is implied.

-q Generate full source-level debugging information to allow the compiled program to be debugged with the TI **FLASH** Studio source-level debugger. Optimizations that would confuse a source-level debugger are disabled when the **-q** flag is specified.

-q1 Generate full source-level debugging information for everything except automatic variables and allow the compiler to operate with all optimizations enabled. When debugging with the **-q1** flag, the locations (register or stack offset) of automatic variables can be easily determined by issuing the TI **FLASH** Studio low-level-display command and examining the C statements with the corresponding assembly code intermixed.

-Q Suppress the Sierra Systems copyright notice when the compiler is invoked.

-s Preprocess and parse the C source file, but do not generate an assembly language output. This option should be used to save time during initial compilations when syntax errors are expected.

-T Process ANSI trigraphs, e.g., map ??(to [and ??) to].

-u Print command line usage information.

-U *name* Undefine the name *name* that was previously defined on the command line using the **-D** flag.

-X Turn on the following specified compiler options.

-Z Turn off the following specified compiler options.

A The **-XA** flag causes address pointers to be returned in register **d0** instead of register **a0**.

b Warn about automatic variables that may be defined before they are used. The compiler will erroneously issue a warning if it finds a path that would result in a use before definition when such a path will never be taken during program execution.

c Make a function call to **__line_ck** immediately before every line of C source. The **__line_ck** function can be used to perform user-supplied debugging functions.

C*length* Set the maximum length of character strings to *length*. The default length is 512 characters. The maximum length that can be specified is limited by available memory. This option is not valid with the **-Z** flag.

d Reference data objects relative to address register **a5** with a 16-bit displacement.

e Write compiler errors to the file *infile*.err where *infile* is the base name of the compiler input file. Errors are also written to **stderr**.

E*size* Set the *size* of the expression stack. The default size of the expression stack is 30. This option is not valid with the **-Z** flag.

f# Generate a warning when a function is declared without prototype information and/or a function call is made in the absence of any function declaration. **f1** causes a warning to be issued when a function is called outside the scope of function declaration and **f2** causes a warning to be issued when a function is declared without prototype information. **f3** specifies the actions of both **f1** and **f2** — i.e., a warning is issued when a function is called without a prototype declaration in scope.

i Not supported by Texas Instruments, however, **i** is recognized as a reserved flag by the compiler.

I Define type **int** to be 16 bits instead of 32 bits. When an **int** is defined to be 16 bits, variables of type **long** (or cast to type **long**) must be used when indexing into an array larger than 32767 bytes or when offsetting a pointer to an object larger than 32767 bytes. Both the TI-89 and TI-92 Plus assume this option is selected and all library functions support 16-bit integers.

l Use long offsets in table look-up switch tables instead of word offsets. Long offsets must be specified if the total size of the code inside a switch statement exceeds 32767 bytes.

L Reference data objects using the 32-bit absolute addressing mode. This option inhibits data references from being coerced to PC-relative modes when the generation of position-independent code is specified (**-Xp** flag). See also **-XW** flag.

M Not supported by Texas Instruments, however, **M** is recognized as a reserved flag by the compiler.

N Automatically size enumerated data types to the smallest integer type that can represent all associated enumeration members. When this option is not specified, enumerated data types are always type **int**.

p Generate position-independent code with a maximum PC-relative displacement of 16 bits.

q When used with the **-Z** flag, this option undoes the effect of the **-q** and **-q1** flags used to specify the generation of source-level debugger information. This option is not valid with the **-X** flag.

r# Prevent the compiler from using the registers specified by the **r** options. The reserved registers are available for use (possibly as base registers) in assembly language routines that are linked with C programs. The options **r0** through **ra** cause the following registers to be reserved:

r0 – **a5**
r1 – **a5, a4**
r2 – **d7**
r3 – **d7, d6**
r4 – **d7, d6, d5**
r5 – **a5, d7**
r6 – **a5, a4, d7**
r7 – **a5, d7, d6**
r8 – **a5, a4, d7, d6**
r9 – **a5, a4, d7, d6, d5**
ra – **a5, a4, a3, d7, d6, d5**

If register **a5** is already being used as a base register as specified by **-Xd**, the reserved address registers are shifted down one register (e.g., **a5**, **a4** becomes **a4**, **a3**).

R Base the relative path of **#include** files on the directory containing the original source file. The default is to base the path on the directory containing the including file. This option is only valid with the **-Z** flag and is provided to undo the effect of the **-XR** flag.

s[*size*] Make a function call to __**stk_ck** at the beginning of every function that requires more than *size* bytes of stack space or calls other functions. The needed stack space is available to __**stk_ck** in register **d0**; the default value for the optional stack size is 40. The stack size *size* cannot be specified with the **-Z** flag.

S Make a function call to __**stk_ck** at the beginning of every function. The amount of stack space needed by the function is available to __**stk_ck** in register **d0**. See section **2.12.3 Debugging Functions** for more information.

u Do not place a leading underscore (_) in front of global variables in the generated assembly code. A leading underscore is placed in front of compiler-generated local labels (_L*xxx*) instead.

w# **w1** disables warnings concerning minor problems such as unnecessary assignments and statements that cannot be reached. **w2** disables warnings concerning the compile-time initialization of address register relative data. **w4** causes the compiler to return a nonzero exit code if any warnings are generated. The codes to the **-Xw#** flag must be combined with a bitwise OR operation to disable more than one class of warnings (e.g., when **-Xw1** is followed by **-Xw4** the effect of **-Xw1** is lost, whereas when **-Xw5** is specified the actions of both **-Xw1** and **-Xw4** are in effect).

W Reference data objects using the 16-bit absolute addressing mode. This option can be used when it is known that all data references will be in the top or bottom 32K bytes of memory. It also inhibits data references from being coerced to PC-relative modes when the generation of position-independent code is specified (**-Xp** flag). See also **-XL** flag.

1 Do not generate source-level debugging **.def – .endef** pairs for function definitions. The **.def** – **.endef** pairs for function definitions are generated by default, even in the absence of the **-q** flag.

2 Save and restore register **d2** in all functions that use the register to guarantee that the value of the register remains unmodified across function calls. The default is to use register **d2** as a temporary scratch register with the value not guaranteed across function calls.

2.4. Pragma Directives

A pragma is a preprocessing directive with the following form:

```
#pragma preprocessing_tokens
```

The functionality of a **#pragma** directive is similar to that of a command line flag. The specified behavior, however, can be embedded into the source file and turned on and off multiple times during a single compilation.

The Sierra C™ compiler currently supports five **#pragma** directives. Two of the pragmas allow the default names of the text and data sections to be changed to arbitrary names. One pragma permits integer functions written in C to be used as interrupt handlers. Finally, two pragmas allow selected data objects to be referenced using the absolute long addressing mode when **a4**-relative addressing is selected on the command line. The five **#pragma** directives are described as follows:

#pragma tsection *section_name*

The **#pragma tsection** directive causes data, instructions, and literal strings that will be placed in the text section to be placed in a section with the name *section_name*. The **#pragma tsection** directive can be used multiple times to change to different names or switch back and forth between a pair of names. For example, the directive **#pragma tsection .text** will cause the name of the text section to switch back to **.text**, the default name.

#pragma dsection *section_name*

The **#pragma dsection** directive causes data, instructions, and literal strings that will be placed in the data section to be placed in a section with the name *section_name*. The **#pragma dsection** directive can be used multiple times to change to different names or switch back and forth between a pair of names. For example, **#pragma dsection .data** will cause the name of the data section to switch back to **.data** the default name.

#pragma interrupt

Allow functions written in C to be used as interrupt handlers. When the **#pragma interrupt** directive is placed before a function definition, that function saves and restores the data and address scratch registers (i.e., **d0**, **d1**, **d2**, **a0**, and **a1**), and exits using an **rte** instruction instead of an **rts**. The **#pragma interrupt** directive applies only to the next function and must be repeated to affect additional functions.

#pragma +abs_data

Allow data objects declared in a file that is compiled to access data relative to the program counter or address register **a5** to be accessed using the absolute long addressing mode. All data objects declared between the **#pragma +abs_data** and **#pragma -abs_data** directives will be accessed using the absolute long addressing mode.

#pragma -abs_data

Undo the effect of the **#pragma +abs_data** directive.

#pragma fp_interrupt

Not supported by Texas Instruments.

2.5. Translation Limits

C programs must not include constructs that exceed the following limits:

- 50 nesting levels for **#include** files
- 16 nesting levels for conditional source inclusion
- 20 nesting levels for **switch** statements
- 25 nesting levels for loop statements
- 20 nesting levels for function calls
- 32 nesting levels for macro invocations
- 255 significant initial characters in an identifier (internal and external)
- 255 significant initial characters in a macro name
- 255 parameters in one function definition and call
- 31 parameters in one macro definition and invocation
- 512† characters in a character string (after concatenation)

† The number of characters can be increased with the **-xc** flag.

2.6. Reserved Keywords

Following is the list of reserved C language keywords that the Sierra C compiler recognizes:

asm	**double**	**int**	**struct**
auto	**else**	**long**	**switch**
break	**enum**	**register**	**typedef**
case	**extern**	**return**	**union**
char	**float**	**short**	**unsigned**
const	**for**	**signed**	**void**
continue	**goto**	**sizeof**	**volatile**
default	**if**	**static**	**while**
do			

2.6.1. ASM Keyword

The **asm** keyword is a standard extension to ANSI C that allows statements to be inserted directly into compiler-generated assembly code. The **asm** keyword is followed by a character string enclosed in parentheses, and it can appear in the C source file wherever a statement or declaration is allowed. For example, the following statement causes the instruction enclosed in double quotes to be inserted into the assembly output at the location that corresponds to the **asm** statement's location in the C source:

```
asm("move.w #0x2400, sr", 4)
```

The integer constant 4 that follows the string indicates the size of the instruction in bytes. The *size* argument is optional; if omitted, the instruction is assumed to be two bytes long. It is necessary to specify the size of the instruction only when the **asm** statement appears inside a loop, **if** statement, **switch** statement, or between a **goto** statement and the label it references. An incorrect or missing size specification may cause the compiler to generate an improperly sized branch instruction, which will result in an assembly error.

2.6.2. ANSI C Keywords

The ANSI C standard added three keywords, **const**, **volatile**, and **signed**, to the C language. The **const** and **volatile** keywords are extremely important to embedded systems developers. These two keywords are fully described in sections **2.9.8 Const Type Specifier** and **2.9.9 Volatile Type Specifier**. The keyword **signed** aids in writing portable code. For example, some compilers recognize objects of type **char** as signed and others recognize them as unsigned. The keyword **signed** can be used in conjunction with the previously existing keyword **unsigned** to control the interpretation of certain object

declarations. The Sierra C compiler treats bit fields as unsigned objects by default. The **signed** keyword can be used to treat bit fields as signed objects.

2.7. Constants

The C compiler recognizes four types of constants: floating-point, integer, character, and enumeration. Every constant has a type that is determined by its form and value, as described in this section. All constants are non-negative in the absence of overflow. If there is a minus sign preceding a constant, it is recognized as a unary operator applied to the constant, and not as part of the constant itself.

2.7.1. Floating-Point Constants

A floating-point constant may be written with a decimal point, a signed exponent, and/or a type specifier suffix. The exponent consists of the letter **e** or **E** followed by an optionally signed decimal constant. The type specifier is one of the letters **f**, **F**, **l**, or **L**. Either the decimal point or the exponent must be present for the constant to be recognized as a floating-point type.

Examples of floating-point constants include:

```
0.          5e9          3.1415
1.0         5E+9l        0.31415E1F
.0          5e+9L        31415.e-4
.1          .50e-8       31415e-4f
```

The digit sequences are interpreted as decimal numbers. The exponent indicates the power of 10 by which the value to the left of the exponent is to be multiplied.

> **Note:** All floating-point constants, whether they are written with a suffix or without a suffix, are of type **double** (10 byte TI BCD floating point).

2.7.2. Integer Constants

An integer constant begins with a decimal digit and contains no decimal point or exponent. It can have a prefix that specifies its base and a suffix that specifies its type. The type specifiers are the letters **u**, **U**, **l**, and **L**, where **u** (or **U**) and **l** (or **L**) can be used together in either order. The **u** (or **U**) specifier coerces the constant to an **unsigned** type, and the **l** (or **L**) specifier coerces the constant to a **long int**.

An integer constant can be represented in three different bases: octal, decimal, and hexadecimal. A constant's base is determined by its leading character(s). Octal constants begin with a zero, and hexadecimal constants begin with the character pair 0x or 0X. Decimal constants begin with any nonzero digit.

Examples of integer constants include:

```
15092        0xf9a        0XF9al
03510        275uL        319LU
```

The type of an integer constant is determined by its value, base, and suffix (if any) according to the following rules:

1. The type of a decimal constant with no suffix is the first type in the following list in which its value can be represented: **int**, **long int**, **unsigned long int**.
2. The type of an octal or hexadecimal constant with no suffix is the first type in the following list in which its value can be represented: **int**, **unsigned int**, **long int**, **unsigned long int**.
3. The type of a constant with only the **u** (or **U**) suffix is the first type in the following list in which its value can be represented: **unsigned int**, **unsigned long int**.
4. The type of a constant with only the **l** (or **L**) suffix is the first type in the following list in which its value can be represented: **long int**, **unsigned long int**.
5. The type of a constant with both the **u** (or **U**) suffix and the **l** (or **L**) suffix is **unsigned long int**.

For example, the constant 0x81234567 is recognized as an **unsigned long int**, because it cannot be represented as a 32-bit **signed long int**.

2.7.3. Enumeration Constants

An identifier declared as an enumeration constant has type **int**. An enumeration constant can be used anywhere an integer constant is allowed; however, there are some restrictions when an enumeration constant is assigned to an enumeration variable. See section **2.9.5 Enumeration Types**, for additional information.

2.7.4. Character Constants

A character constant is a sequence of up to four characters enclosed in single quotes ('). Characters that cannot be entered directly or conveniently into the source program, such as nongraphic characters, can be specified in a character constant using an escape sequence. Table 2.1 lists the available escape sequences and their values.

Character constants have type **int**. The value of each character in a character constant is its integer encoding in the ASCII character set or the value of its associated escape sequence, whichever is applicable.

The double quote (") and the question mark (?) can be specified directly or with their escape sequences. The single quote (') and the backslash (\), however, must be specified with their escape sequences.

In the octal and hexadecimal escape sequences, *ooo* represents up to three octal digits and *hhh* represents up to three hexadecimal digits. These digit sequences are terminated with the third digit or the first nonoctal (or nonhexadecimal) character. The specified digits determine the value of the character.

Escape Sequence	Common Name	Integer Value
\a	alert (bell)	0x7
\b	backspace	0x8
\f	form feed	0xC
\n	newline	0xA
\r	carriage return	0xD
\t	horizontal tab	0x9
\v	vertical tab	0xB
\'	single quote	0x27
\"	double quote	0x22
\?	question mark	0x3F
\\	backslash	0x5C
ooo	octal escape code	0*ooo* (masked with 0xFF)
\x*hhh*	hexadecimal escape code	0x*hhh* (masked with 0xFF)

Table 2.1: Escape Characters

Some integer values that can be represented using three octal or hexadecimal digits will not fit into a single eight-bit character. When a value does not fit into eight bits, it is masked with the value 0xFF to force it to fit into eight bits. The construct ' \x123 ' is a character constant containing a single character. To

specify a character constant containing the two characters whose values are ' \12 ' and ' 3 ', the construction ' \0123 ' must be used. To specify a character constant containing the two characters whose values are ' \12 ' and ' z ', the constructions ' \012z ' and ' \12z ' are both valid.

The sign of the character constant is determined by the most significant bit of the character. For example, the character constant ' \xFF ' has the value -1, and the character constant ' \x7F ' has the value +127. The value of a multi-character character constant is determined by placing its characters (from right to left) into successively higher order bytes of an integer (beginning with the lowest order byte). Just as the sign of a single-character character constant is determined by the value of bit 7, the sign of a four-character character constant is determined by the value of bit 31. Two- and three-character character constants are always positive.

The examples in Table 2.2 illustrate how character constants are evaluated.

Character Constant	Hexadecimal Value	Decimal Value
'\0'	0x0	0
'0'	0x30	48
'\x123'	0x23	35
'\x50'	0x50	80
'\x90'	0x90	-112
'\x08012'	0x803132	401202
'1234'	0x31323334	825373492
'\x040123'	0x40313233	1076965939
'\x080123'	0x80313233	-2144259533

Table 2.2: Character Constants

2.8. Character Strings

A character string is a sequence of zero or more characters enclosed in double quotes (e.g., "abc"). The same considerations that apply to characters and escape sequences in character constants apply also to character strings.

Refer to Table 2.1 for the list of recognized escape sequences and the rules and restrictions that apply. In a character string, the only difference in the use of the escape sequence mechanism is that the single quote (') can be specified directly or with its escape sequence, whereas the double quote (") must be specified with its escape sequence.

A character string is actually a static array of characters, where the array has been initialized with the given characters and a terminating null character. Character strings cannot be modified during program execution; the static array containing the characters resides in the program text section, which is typically loaded into Read-Only Memory (ROM).

A newline is illegal inside of a character string. There are two ways, however, to continue a character string on a new line. A character string may be continued by placing a backslash (\) immediately before the newline. For example, the following two strings are equivalent:

```
"this is a test of a string that \
spans two lines"

"this is a test of a string that spans two lines"
```

Character strings can also be continued by placing them adjacent to each other. For example, the following two strings are equivalent:

```
"this" " string "    "has "
"been broken"" into several pieces"

"this string has been broken into several pieces"
```

Escape sequences are converted into single characters before adjacent character strings are concatenated. For example, the first string below is equivalent to the second string, not the third:

```
"\x12"  "3"

"\x0123"     (two characters)

"\x123"      (one character)
```

The maximum length of a character string is 512 characters unless it is increased with the **-XC** flag. When two or more strings are concatenated, the character length limit applies to the combined length.

2.9. Types and Representations

This section describes the internal representations of the various integer and floating-point types. It also describes the correct usage of enumeration types and bit fields, and discusses the issues concerning the **const**, **volatile**, and **void** type specifiers.

2.9.1. Integer Types

Table 2.3 summarizes the recognized integer types, their sizes, and their value ranges. Integer data types can also be declared to be **signed** just as they can be declared to be **unsigned**. The **signed** keyword, however, is significant only when used in a bit field declaration, because bit fields are the only integer data types that are unsigned by default.

The integer data types **int** and **short int** declare objects of the same size (when the **`-XI`** command line flag is used to specify 16-bit integers as on the TI-89 / TI-92 Plus). Even though objects of both types have the same internal representation and can be used to store the same values, the two types should not be used interchangeably. One reason is that pointers to **int** and pointers to **short int** are different types, and a warning will be issued if either type is assigned to the other. Also, the program may someday be recompiled on another compiler where **int** and **short int** do not declare objects of the same size.

Type	Size in Bits	Minimum Value	Maximum Value
char	8	-128	127
short int	16	-32768	32767
int †	16	-32768	32767
long int	32	-2147483648	2147483647
unsigned char	8	0	255
unsigned short int	16	0	65535
unsigned int†	16	0	65535
unsigned long int	32	0	4294967295

† Integers are 16-bit objects on the TI-89 / TI-92 Plus (the **`-XI`** command line flag is specified).

Table 2.3: Integer Types

2.9.2. Integer Representations

The memory address of any integer data object is the address of its highest order byte (i.e., the byte containing the most significant bits). Unsigned integer data types are represented as straight binary numbers, and signed integer types are represented using two's complement notation. Figure 2.1 shows the internal representations of the three basic integer data types for the given values.

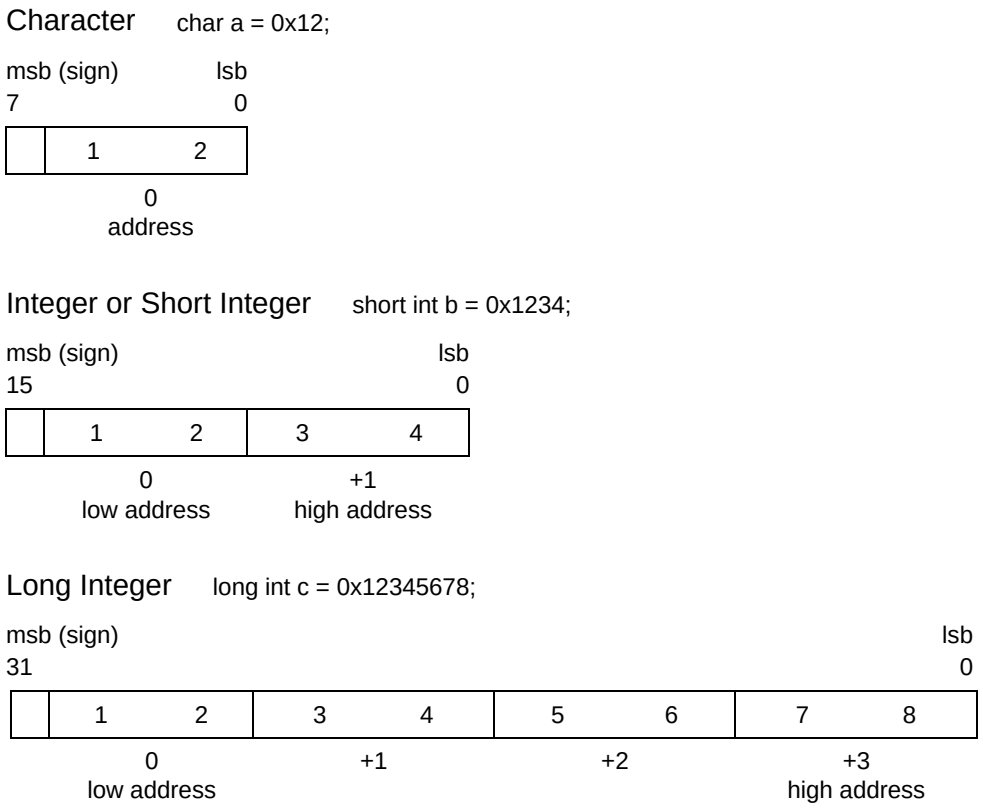


Figure 2.1: Internal Integer Representations

2.9.3. Floating-Point Types

The Sierra C compiler has been modified by Texas Instruments to support the TI BCD floating-point format, removing all other previously supported floating-point formats and coprocessors, except for an IEEE format which is used in special cases and exists only in the compiler, not on the TI-89 / TI-92 Plus. (See section **2.12.2 Internal Floating-Point Functions** for more information on the IEEE format.)

There are two forms of the TI BCD floating-point values, differing only in the number of significant digits retained in the mantissa. Both have a size of ten bytes, with **double** keeping 16 BCD digits in the mantissa, while **float** rounds the accuracy to 14 digits. There is not a separate representation for **long double**, which is interpreted as **double** when encountered. The floating-point values available to the users of the TI-89 and TI-92 Plus calculators have 14 digit mantissas. It is highly recommended to always use the **double** format when coding the applications to take advantage of the increased accuracy provided by the extra two digits. See chapter **16. Working with Numbers** in the TI-89 / TI-92 Plus Developers Guide for information on how and when to round to the 14 digits required by the calculator user floating-point representation. The **float** format will not usually be used in TI-89 / TI-92 Plus applications. In the

include file (**tiams.h**) supplied with the TI-89 / TI-92 Plus SDK, BCD16 has been defined as **double**, and should be used when coding applications as a reminder that routines containing floating-point operations may not be portable code.

2.9.4. Floating-Point Representations

All TI BCD floating points are 10 byte objects. The first two bytes in the TI BCD floating-point format are the mantissa sign and exponent. The mantissa sign is the most significant bit of that word (1=negative, 0=positive) and the other 15 bits are a 0x4000 biased exponent, where a value less than 0x4000 represents a negative exponent, and a value greater than 0x4000 represents a positive exponent. The memory address of a floating-point data object is the address of the first byte of the sign/exponent. The decimal point is assumed to be after the first BCD mantissa digit. The mantissa consists of 16 BCD digits, with digits 15 and 16 always equal to 0 in a **float**.

double and float

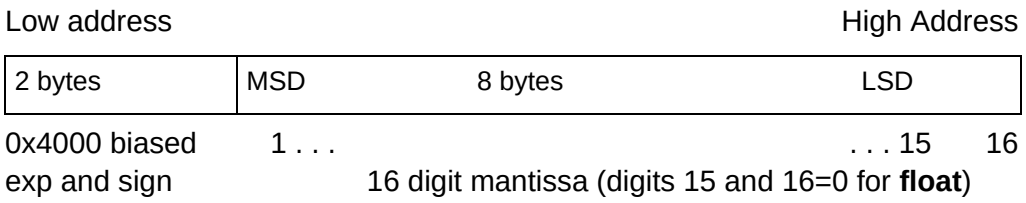


Figure 2.2: Internal TI BCD Floating-Point Representation

Following are some examples of common floating-point values shown as if written in 68000 assembly language:

```
_PI:
   .word   0x4000              ; 3.141592653589793
   .long   0x31415926,0x53589793
_FPZERO:
   .word   0x4000              ; 0
   .long   0,0
_FPPT001:
   .word   0x3FFD              ; .001
   .long   0x10000000,0
_FPPTFIVE:
   .word   0x3FFF              ; .5
   .long   0x50000000,0
_FPONE:
   .word   0x4000              ; 1
   .long   0x10000000,0
_FPNEG1:
   .word   0xC000              ; -1
   .long   0x10000000,0
_FPTEN:
   .word   0x4001              ; 10
   .long   0x10000000,0
```

```
_FPNEGTEN:
   .word   0xC001              ; -10
   .long   0x10000000,0
```

The internal floating-point representation allows an exponent range of -16384 to 16383. However, the exponent range available to the users of the TI-89 and TI-92 Plus calculators is -999 to 999. See chapter **16. Working with Numbers** in the TI-89 / TI-92 Plus Developers Guide for information on how and when to verify that the exponent is within the range required by the calculator user floating-point representation and what to do when it is not.

There are also several specific floating-point representations for signed zeros, infinities, and undefined values (or NANs).

positive zero

Low address High Address

2 bytes	MSD 8 bytes LSD
0x0000	0x00000000, 0x00000000

negative zero

Low address High Address

2 bytes	MSD 8 bytes LSD
0x8000	0x00000000, 0x00000000

positive infinity

Low address High Address

2 bytes	MSD 8 bytes LSD
0x7FFF	0xAA00BB00, 0x00000000

negative infinity

Low address High Address

2 bytes	MSD 8 bytes LSD
0xFFFF	0xAA00BB00, 0x00000000

unsigned infinity

Low address High Address

2 bytes	MSD 8 bytes LSD
0x7FFF	0xAA00CC00, 0x00000000

invalid floating-point representation (undefined or NAN)

Low address High Address

2 bytes	MSD	8 bytes	LSD
0x7FFF	0xAA000000, 0x00000000		

Figure 2.3: Special Internal Floating-Point Representations

All of the special internal floating-point values are valid inputs to TI BCD floating-point routines and will be handled correctly when encountered. It is not necessary to check for them after every floating-point operation to detect overflows or other special values. However, none of the special values shown above can be directly entered on the TI-89 or TI-92 Plus calculators. The signed zeros may result from a calculation but since they are displayed as 0. on the calculator, they can only be recognized by their behavior in other calculations. The infinities and invalid floating-point values must be converted to other representations before being made available to the user. See chapter **16. Working with Numbers** in the TI-89 / TI-92 Plus Developers Guide for information on how and when to verify that the floating-point value is valid as required by the calculator user floating-point representation and what to do when it is not.

2.9.5. Enumeration Types

An enumeration type is a set of integer values represented by identifiers; these identifiers are referred to as enumeration constants. The declaration of an enumeration type is similar to that of a structure or union type. For example, the following declaration creates an enumeration type **color** — whose values are **red**, **green**, **blue**, and **white** — and declares the variables **boat** and **car** to be of this type:

```
enum color { red, green, blue = 8, white } boat, car;
```

The enumeration constants (**red**, **green**, **blue**, and **white** in the above example) are assigned integer values when the **enum** type is created. Enumeration constants are assigned either values generated automatically by the compiler or values specified by an assignment operator and a constant integer expression. When an explicit assignment is not made, the first enumeration constant in the list is assigned the value zero; subsequent enumeration constants are assigned the value of the previous constant plus one. In the above example, the enumeration constants have the following values:

```
red = 0
green = 1
blue = 8
white = 9
```

Using the enumeration constant assignment capability, it is possible (and legal) to create different enumeration constants with the same integer values.

Enumeration constants are integer type objects and are treated the same as numerical integer constants (i.e., their values cannot be modified). They can be used anywhere an integer constant can be used except in an assignment to an enumeration variable of a different **enum** type. Enumeration constants within the same block scope must have unique names so that they can be distinguished from each other as well as other variables, functions, and **typedef** names.

Variables and other objects of enumeration type can be declared in the declaration containing the type definition (as in the above example) or in subsequent declarations that specify the enumeration type name. If subsequent declarations will not be made, the enumeration type name (**enum** tag) can be omitted from the initial declaration. For example, the following three sets of declarations are equivalent:

```
enum { sunny, overcast, rainy } CA, FL, NJ;

enum weather { sunny, overcast, rainy } CA, FL, NJ;

enum weather { sunny, overcast, rainy };
enum weather CA, FL;
enum weather NJ;
```

The **enum** tag (**weather** in the above example) is placed into the same name space as **struct** and **union** tags.

Enumeration variables and objects are treated the same as integers except in assignments that involve incompatible enumeration types. An enumeration type is the same size as an **int** unless the **-XN** flag is specified. When this flag is specified, the size of an enumeration type is determined by the values of its associated enumeration constants. If the value of each enumeration constant fits in a **signed char**, the enumeration type is the same size as a **char**. If the value of each enumeration constant fits in a **signed short int**, the enumeration type is the same size as a **short int**. Otherwise, the enumeration type is the same size as an **int**.

As previously stated, enumeration constants and variables can be used anywhere integer types can be used except in assignments involving different enumeration types. The following examples utilize the enumeration objects declared on the previous page to demonstrate **enum** type checking across assignments:

```
int i;

CA = sunny;       /* legal                          */
i = 25;           /* legal                          */
red = 36;         /* illegal, red is constant       */
car = boat;       /* legal                          */
car = NJ;         /* illegal                        */
FL = green;       /* illegal                        */
boat = i;         /* legal                          */
```

2.9.6. Bit Field Description

The C language allows integer data to be stored in spaces that differ in size from those provided by the basic integer types. Such arbitrarily sized integers are called bit fields. They are supported by allowing the width in bits of a structure or union member to be specified using a colon and a constant expression after the member declarator. In the following declaration, x.a is a 10-bit integer, x.b is a 12-bit integer, and x.c is a seven-bit integer:

```
struct s {
       long int a:10;
       long int b:12, c:7;
} x;
```

The three members of this structure occupy a total of four bytes; internally, they are packed into a space that has the same size and alignment as a **long int**.

Bit fields can be packed in any of the four basic integer types (for example, **char**, **short int**, **int**, or **long int**). The size of the type determines the maximum allowable bit field size. Bit field declarations can also include the **signed** and **unsigned** type specifiers. The **signed** type specifier causes the bit field to be interpreted as a signed quantity — the most significant bit of the bit field (for example, the sign bit) is extended when extracting the contents of the field. The **unsigned** type specifier can also be used, but has no effect since bit fields are unsigned by default. Given the following initialization, the three bit field members will have the values shown:

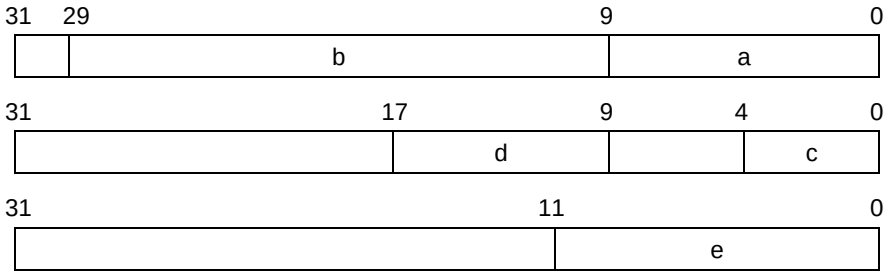
```
struct t {
       int i:3;
       signed int j:3;
       unsigned int k:3;
} y = { -1, -1, -1 };      /* -1 bit pattern all 1's */

y.i = 7
y.j = -1
y.k = 7
```

Note: Since most computers do not allow bits to be addressed directly, it is illegal to take the address of a bit field. As a result, pointers to bit fields and arrays of bit fields are not permitted. Furthermore, functions are not allowed to return bit fields.

2.9.7. Bit Field Internal Representation

Bit fields are packed into memory as efficiently as possible. Each bit field is placed in the lowest order bits (highest address) available in the specified integer type. If a bit field does not fit in the remainder of the current integer, it is placed in a new one (i.e., it cannot span two integers). A bit field declaration without an identifier is used to force a desired alignment. If the field width is nonzero, the bits are allocated exactly as if an identifier was present in the declaration; if the field width is zero, the remainder of the current integer is skipped. For example, the following declaration causes memory to be allocated as shown:

```
struct tag {
       long int a:10;
       long int b:20;
       long int c:5;
       long int :5;
       long int d:8;
       long int :0;
       long int e:12;
       } s;
```



2.9.8. Const Type Specifier

The value of an object that is declared with the **const** type specifier cannot be modified; therefore, the object cannot appear as the left-hand side of an assignment, nor as the operand of the '++' or '– –' operator. The following examples demonstrate the legality of various modifications involving **const** objects:

```
const int a = 99;
const int b = 66;
const int *p = &a;

a++;                    /* illegal     */
b--;                    /* illegal     */
a = 10;                 /* illegal     */
p = &b;                 /* legal       */
*p = 20;                /* illegal     */
```

The **const** type specifier can be used with integer types, floating-point types, structure and union types, and enumeration types. It can also be used with individual members of structures and unions, in which case only the specified members are affected. If the **const** type specifier is used without any other type specifiers or with only the **volatile** type specifier (see below), type **int** is implied.

The **const** type specifier can also be used to declare pointers to constant objects and constant pointers to constant and nonconstant objects. Just as an asterisk (*) in a declaration indicates a pointer to an object, the construct ***const** indicates a constant pointer to an object. A constant pointer cannot be modified; no restrictions are placed on the object to which it points. The following declarations define a constant **int**, a constant pointer to an **int**, and a constant pointer to a pointer to a constant pointer to a constant **double**, respectively:

```
const int a;
int * const b;
const double * const ** const c;
```

A pointer to a nonconstant object can be assigned to a pointer to either a constant or nonconstant object; however, a pointer to a constant object can only be assigned to a pointer to a constant object. These pointer assignment rules were established to help prevent accidental modifications of constant objects with dereferenced pointers to nonconstant objects. These rules can be easily sidestepped with a type cast, but this is not recommended since it could result in an attempt to modify data stored in Read-Only Memory (ROM).

In most embedded systems, as much information as possible is placed in ROM (e.g., the entire code (**.text**) section). The **const** type specifier is extremely important in these environments because it causes declared data (except when also declared **volatile**) to be placed in the **.text** section — i.e., directly in ROM. For additional information on the use of the **const** type specifier, refer to section **2.14 Static Storage Initialization**.

2.9.9. Volatile Type Specifier

An object that is declared with the **volatile** type specifier is guaranteed to be accessed, in its entirety, each time it is referenced in the source. As a result, the compiler must forgo optimizations that would alter either the size of references to the object (e.g., **char**, **short**, **int**, etc.) or the number of references made to the object. For example, the compiler cannot test a bit in a four-byte volatile object by referencing only the byte that contains the bit of interest; the object must be referenced as a four-byte entity. As the following example illustrates, the compiler would not be able to optimize the first function, which reads data from an I/O port, as though it were written as the second function:

```
volatile char * const stat_reg = (char *) 0x60000D;
volatile char * const rcvr_reg = (char *) 0x60000F;

get_char()
{

       while( !(*stat_reg & 0x40) );
       return( *rcvr_reg );
}

get_char()
{
       register int temp;

       temp = !(*stat_reg & 0x40);
       while( temp );                          /* infinite loop */
       return( *rcvr_reg );
}
```

The **volatile** type specifier is used for memory-mapped I/O ports, variables shared by other processes, variables modified inside interrupt handlers and error handlers, and any other variables that are accessed in ways not obvious to the compiler.

The **volatile** type specifier observes the same declaration usage rules as the **const** type specifier (see section **2.9.8 Const Type Specifier**); it can be used in conjunction with the **const** type specifier. The following declarations define a volatile **int**, a constant volatile **int**, and a volatile pointer to a constant volatile pointer to a pointer to a volatile **double**, respectively:

```
volatile int a;
volatile const int b;
volatile double ** volatile const * volatile c;
```

The second declaration above could be used for a resource such as a real-time clock; the compiler does not know when the value of a real-time clock changes, and a program should not attempt to modify it.

A pointer to a nonvolatile object can be assigned to a pointer to either a volatile or nonvolatile object; however, a pointer to a volatile object can only be assigned to a pointer to a volatile object. These pointer assignment rules, which are analogous to the rules for constant pointers described above, were established to help prevent accidental references of volatile objects with dereferenced pointers to nonvolatile objects.

2.9.10. Touch Operator

The _**touch(***pointer_to_volatile_argument*) operator uses the test instruction, **tst**, to "touch" the memory location pointed to by *pointer_to_volatile_argument*. In embedded systems design, it is often necessary to touch an address-mapped piece of hardware — e.g., to increment a counter or set a latch. The argument to _**touch( )** must be a dereferenced pointer to a **volatile** integer type object; otherwise, _**touch( )** behaves as an ordinary function.

2.9.11. Void Type Specifier

An expression of type **void** has no value associated with it. Such an expression can be neither referenced nor converted (implicitly or explicitly) to another type.

The **void** type specifier is most frequently used in the declaration of functions that return no values. It is illegal for a **void** function to return a value or for the return value of a **void** function to be used in an expression. The **void** type can also be used to indicate that the value of an expression is ignored. For example, the following function, which returns no value, uses a **void** cast to discard the return value of the **printf** function:

```
void report_error(int error, int line)
{
       (void)printf("Error %d on line: %d\n", error, line);
}
```

The **void** cast is not necessary in the above example, but many programmers use the cast to indicate that the return value is being intentionally ignored.

2.9.12. Void Pointer (void *)

A pointer to **void** (i.e., **void ***), referred to as a generic pointer, has special meaning to the compiler. A pointer to any type of object can be converted to a **void** pointer and back again with the resultant and original pointers guaranteed to compare equal. Additionally, **void** pointers and non-**void** pointers may appear together in assignment and comparison expressions with no explicit type conversion. The following examples demonstrate legal and illegal pointer expressions:

```
int *pi;
char *pc;
void *pv;

pv = pc;     /* legal                                    */
pc = pi;     /* illegal, cast required                   */
pi++;        /* legal                                    */
pv += 2;     /* illegal, size of void pointer unknown    */
f( *pv )     /* illegal to dereference a void pointer    */
```

2.10. Conversions

The compiler provides for both the implicit and explicit conversion of values from one type to another.

- A value may be explicitly converted to another type with a cast operator.
- An operand may be implicitly converted to another type so that a specific arithmetic or logical operation can be performed.
- An implicit conversion may result from the assignment of an object of one type to an object of another type.
- An argument to a function may be implicitly converted to another type in preparation for the function call.
- The value returned by a function may be implicitly converted to the function return type before the function return.

2.10.1. General Considerations

When a value of one type is converted to a value of another type, the internal representation (i.e., bit pattern) may change. Conversions between floating-point types and integer types always involve a change in representation. When

converting between different integer types, the resultant representation is easily determined since the target machine uses a two's complement representation. Conversions between integer types of the same length involve no change in representation. Conversions from longer integer types to shorter integer types involve a truncation of the excess high order bits. Conversions from shorter integer types to longer integer types involve the padding of the additional high order bits with all 1's or 0's.

2.10.2. Integer Types

When an integer of one type is converted to an integer of another type, the resultant value is determined according to the following rules:

- When a signed integer type is converted to another signed integer type of equal or greater length, the value remains unchanged.
- When an unsigned integer type is converted to another unsigned integer type of equal or greater length, the value remains unchanged.
- When an unsigned integer type is converted to an integer type of greater length, the value remains unchanged.
- When an integer type is converted to an integer type of shorter length, the resultant value is the value of the truncated bit pattern as interpreted by the new type.
- When a signed integer type is converted to an unsigned integer type of greater length the resultant value is unchanged if the value of the signed integer was non-negative. Otherwise, the bit pattern representation in the unsigned integer is determined by promoting the signed integer to a signed integer of the same length as the unsigned integer.
- When an integer type is converted to an integer type of the same length, the resultant value is the value of the bit pattern as interpreted by the new type.

2.10.3. Floating-Point and Integer Types

When a floating-point type is converted to an integer type, the resultant value is determined by discarding the fractional part. If the conversion is to an **unsigned long** and the value of the integral part cannot fit in the space provided, the behavior is undefined. If the conversion is to any other integer type and the value of the integral part cannot fit, the integral part is first assigned to a **signed long** and is then converted to the specified integer type using the integer conversion rules. If the integral part cannot fit in a **signed long**, it is assigned the largest positive or negative value (depending on the sign of the integral part) that can be represented by a **signed long**.

2.10.4. Floating-Point Types

Since all TI BCD floating-point data objects are ten bytes, there is no promoting **float** to **double** or demoting **double** to **float**. Explicit type casting of **float** to **double** or **double** to **float** is also ignored. This does not imply that there is no difference between them. TI floating-point arithmetic will recognize that a **float** has only 14 significant digits. If a cast from **double** to **float** is desired, the **round14** function on the TI-89 / TI-92 Plus is available. However, it is strongly recommended to always use **double**, taking advantage of the increased accuracy. (See section **2.9.3 Floating-Point Types** for more information.)

2.10.5. Usual Arithmetic Conversions

Most binary operators that require operands of arithmetic type cause implicit conversions (to yield a common type). These conversions are known as the *usual arithmetic conversions*. The first applicable rule in the following lists specifies the performed conversions:

- If either operand is of type **double**, the other operand is converted to type **double**.
- If either operand is of type **float**, the other operand is converted to type **float**.
- If either operand is of type **unsigned long int**, the other operand is converted to type **unsigned long int**.
- If one operand has type **long int** and the other has type **unsigned int**, the conversion depends on the specified integer size (see **`-XI`** flag). When 16-bit integers are used (as on the TI-89 / TI-92 Plus), the operand of type **unsigned int** is converted to the type **long int**; otherwise, both operands are converted to the type **unsigned long int**.
- If either operand is of type **long int**, the other operand is converted to type **long int**.
- If either operand is of type **unsigned int**, the other operand is converted to type **unsigned int**.
- If either operand is *not* of type **int**, it is converted to type **int**. (Exception: if 16-bit integers are used, as on the TI-89 / TI-92 Plus, and the operand type is **unsigned short int**, it will be converted to **unsigned int** in order to preserve its value.)
- If both operands are of type **int**, no conversion is necessary.

> **Note:** These binary conversion rules do not apply to the shift operators (>> and <<), since the operands are not combined directly. Instead, each operand is treated as if it were an operand of a unary operator.

> **Note:** Operands of type integer are not always converted as stated above. Sometimes, smaller integer types are used if doing so results in improved run-time performance without loss of precision.

2.10.6. Restrictions

Type conversions can be performed on any of the scalar types — i.e., integer types, floating-point types, enumeration types, and pointers. There are, however, certain restrictions imposed upon both explicit and implicit conversions.

The only explicit conversions that are not permitted are those between floating-point types and pointers; any other conversions involving scalar types are allowed.

Additional restrictions apply to implicit conversions that are made across assignments and through function returns. Any statement requiring either an implicit conversion between two enumeration types or an implicit conversion involving a pointer (excluding null pointers and void pointers) is illegal. In both of these cases, however, the conversion will be performed and a warning issued.

All conversions between **float** and **double** are ignored since they are both 10 byte objects, however, no warning or error is issued.

2.11. Function Calling Conventions

When a function is called with arguments, the value of each specified argument is pushed onto the stack. (Hereafter, the value of an argument will be referred to simply as the argument.) The arguments are pushed onto the stack in reverse order — i.e., beginning with the rightmost argument. After a function call, the stack pointer is incremented, if necessary, to restore its value to that prior to the stacking of the arguments.

> **Note:** When the **`-0`**, **`-0z`**, or **`-0c1`** command line flag is specified, the stack pointer will not necessarily be incremented after every function call that has arguments pushed onto the stack. In the presence of any of these flags, attempts are made to coalesce more than one stack cleanup into a single stack adjustment.

> **Note:** If the top of the stack is free, the rightmost argument is sometimes moved directly into the top stack position (i.e., the stack pointer is not decremented); otherwise, it is pushed onto the stack. Additional arguments are always pushed onto the stack.

The amount of stack space used by each function argument is determined by the type of the argument and whether the function is declared with a function prototype. Structure arguments are pushed onto the stack and take as much stack space as necessary to fit the entire structure. If necessary, the stack space allocated to a structure argument is rounded up to a multiple of four bytes.

2.11.1. Declarations and Definitions

A function *declaration* declares a variable to be a function and specifies the type of value it returns. A prototype function declaration also establishes the number of function arguments and the types of those arguments. A function declaration does not cause memory to be allocated or code to be generated. A function *definition* causes memory to be allocated and code to be generated; it also serves as a function declaration.

2.11.1.1. Function Prototypes

A function prototype is a function declaration or definition that specifies both the number of arguments and the argument types. Function prototypes prevent errors caused by passing the wrong argument type or wrong number of arguments to a called function.

The following are examples of prototype function declarations:

```
double f1( short x, int y, double z );
double f2( short, int, double );
double f3( short, int, double, ... );
double f4( void );
```

The functions f1, f2, f3, and f4 are declared to return a value of type **double**. The first two functions are declared to accept exactly three arguments: a **short**, an **int**, and a **double**, respectively. Except for the function names, the first two function declarations are identical. The formal parameters x, y, and z in function f1 are optional and are used for documentation purposes only. The scope of these optional parameter names extends only to the end of the declaration; the names do not have to match the formal parameter names in the actual function definition. Function f3 is declared to accept three or more arguments. The ellipsis notation (, . . .) informs the compiler that zero or more additional arguments of unknown type will be passed to the function. In the body of the function definition, the macros **va_start( )** and **va_arg( )** defined in the include file **tiams.h** should be used to access the additional argument values. Function f4 is declared to accept no arguments; calling f4 with any arguments will result in an error.

An in-scope prototype function declaration not only establishes the function return type, but also enables verification that the function is called with the correct number of arguments and that the type of each argument is compatible with its corresponding formal parameter. The type checking and conversions that are performed are identical to those performed when assigning a value using the assignment operator (=). For example, if a function that is defined with a parameter of type **double** is called with an argument of type **int**, the argument is automatically converted to a value of type **double** before being pushed onto the stack; an explicit cast operator is unnecessary. Additional arguments permitted by the ellipsis notation (, . . .) are handled as if they were arguments to a function declared without a prototype. When declared with the ellipsis notation, type checking and automatic type conversions cannot be performed; instead, the ANSI C integral and floating-point promotion rules are applied to the arguments.

The following is an example of a prototype function definition:

```
double f1( short a, int b, double c )
{
       /* function body */
}
```

The function f1 is defined to accept three arguments and return a value of type **double**. The formal parameter type declarations **int a**, **short b**, and **double c** declare how the parameters will be used inside the function. The Sierra C compiler specifies how the parameters are pushed onto the stack at the function call site. If a prototype declaration is in scope, an argument of type **short** may be promoted to an **int** before being pushed onto the stack. The rules for determining how an argument is pushed onto the stack in the presence of a prototype is determined by specific command line flags. Table 2.4 (section **2.11.2 Passing Argument Values**) summarizes how arguments of different types are pushed onto the stack with and without a prototype in scope.

2.11.1.2. Old-Style Declarations

An old-style function declaration is a declaration that provides only the type of the return value. Information on the types and number of arguments is not established. The following is an example of an old-style (nonprototype) function declaration:

```
int f5();
```

The function f5 is declared to return a value of type **int**; no information about the number and types of arguments is specified. This declaration could be deleted from the program without any effect, since the default function return type is **int**.

The following is an example of an old-style function definition:

```
int f5( a, b, c, d, e )
int a;
short b;
long c;
float d;
double e;
{
       /* function body */
}
```

The function f5 is defined to accept five arguments and return a value of type **int**. The formal parameter type declarations **int a**, **short b**, **long c**, **float d**, and **double e** declare how the parameters will be used inside the function. They do not describe the types of the actual parameters that are pushed onto the stack at the function call site. The expected types of the actual parameters are determined by the integral and floating-point promotion rules. The promotion rules state that arguments of type **char** and **short** are converted to type **int**, and that arguments of type **float** are converted to type **double**; no other conversions are performed.

At a function call site, there is no information available on the types of the arguments. If a called function has a formal parameter declared to be of type **float**, it is expecting an actual parameter of type **double** to have been pushed on the stack. If, however, the parameter is of type **int**, it must be explicitly cast to either a **float** or a **double** at the call site to prevent it from being incorrectly pushed as an **int**. Finally, if the function is called with too few or too many parameters, the error will go undetected during compilation.

2.11.1.3. Mixing Prototype and Old-Style Declarations

The Sierra C compiler supports both new-style (prototype) and old-style (nonprototype) function declarations and definitions as specified by the ANSI C standard. However, it is highly recommended that the prototype style be used exclusively.

Mixing prototype declarations and old-style function definitions should be avoided because it will often create problems. The following is an example of a prototype declaration and an old-style function definition:

```
int func( short a );

int func( a )
short a;
{
       /* function body */
}
```

When the file containing the above example is compiled, the following warning will be issued:

```
warning: incompatible parameter types, func() arg 1
```

This message is generated because the prototype declaration declares a function that accepts a **short** parameter, while the function definition declares a function that accepts an **int** parameter. The formal parameter in the function definition is recognized as an **int** because the function definition is in the old-style format; therefore, the integral promotion rules are applied to the **short** parameter. To further understand the problem, examine the case in which a **short** is pushed as two bytes in the presence of a prototype. A two-byte value will be pushed onto the stack because a prototype declaration is in scope; however, a four-byte argument will be expected inside the function that was defined using the old style. A problem also results if a function that is defined with a prototype to accept a **short** is called with an old-style declaration (or no declaration) in scope.

The above problems can be avoided through exclusive use of prototype declarations and definitions. The compiler command line flags **`-Xf1`**, **`-Xf2`**, and **`-Xf3`** can be used to verify that appropriate declarations are being utilized. The **`-Xf1`** flag causes a warning to be issued when a function is called outside the scope of a function declaration. The **`-Xf2`** flag causes a warning to be issued when a function is declared without a prototype. The **`-Xf3`** flag (the combination of **`-Xf1`** and **`-Xf2`**) causes a warning to be issued if a function is called without a prototype declaration in scope.

2.11.2. Passing Argument Values

Several factors determine how an argument value is pushed onto the stack when a function call is made: the specified command line flags, the type of the argument, and whether or not a function prototype is in scope. If a function prototype is not in scope, the integral and floating-point promotion rules will be applied to the argument types.

The **`-XI`** flag instructs the compiler to interpret objects of type **int** as 16 bits rather than 32 bits (default). Specifying the **`-XI`** flag, as on the TI-89 / TI-92 Plus, will cause all integer types (except for type **long**) to be passed as 16-bit objects independent of whether a function prototype is in scope. The **`-0s#`** flags determine how argument values of different types are passed only in the presence of a function prototype. In the presence of a prototype, parameters of type **char** and **short** are pushed as two-byte objects when the **`-0s2`** flag is specified, and are pushed as four-byte objects when the **`-0s4`** flag is specified.

Compiler and Flags	char	short	int	long	float	double
com68	2 / 4	2 / 4	4 / 4	4 / 4	10 / 10	10 / 10
com68 -Os2	2 / 4	2 / 4	4 / 4	4 / 4	10 / 10	10 / 10
com68 -Os3	2 / 4	2 / 4	4 / 4	4 / 4	10 / 10	10 / 10
com68 -Os4	4 / 4	4 / 4	4 / 4	4 / 4	10 / 10	10 / 10
com68 -Os5	4 / 4	4 / 4	4 / 4	4 / 4	10 / 10	10 / 10
com68 -XI	2 / 2	2 / 2	2 / 2	4 / 4	10 / 10	10 / 10
com68 -XI -Os2	2 / 2	2 / 2	2 / 2	4 / 4	10 / 10	10 / 10
com68 -XI -Os3	2 / 2	2 / 2	2 / 2	4 / 4	10 / 10	10 / 10
com68 -XI -Os4	2 / 2	2 / 2	2 / 2	4 / 4	10 / 10	10 / 10
com68 -XI -Os5	2 / 2	2 / 2	2 / 2	4 / 4	10 / 10	10 / 10

Note: Sizes are in bytes, with and without prototype — (with prototype) / (without prototype). The TI-89 / TI-92 Plus requires use of the **-XI** flag.

Table 2.4: Determination of Argument Size

Table 2.4 shows the sizes in bytes that function arguments occupy on the stack. The sizes are shown as a function of the compiler and associated command line flags, argument type, and whether or not a prototype is present. The numbers in the table to the left of the slash (/) are sizes in bytes in the presence of a prototype; the numbers to the right are sizes in the absence of a prototype.

2.11.3. Accessing Parameters

Inside the called function, the function parameters are accessed from the stack. The exact mechanism for accessing the parameters is determined by the presence or absence of the **link** and **movem** instructions.

The **link** instruction, if present, is the first instruction in a function; it is used to set up a stack frame using address register **a6** as the frame pointer (refer to the **link** instruction in the *M68000 Family Programmer's Reference Manual*). The **link** instruction is also used to leave an extra four bytes free on the top of the stack. When the top of the stack is free, the rightmost parameter in a function call can be moved onto the stack instead of pushed onto the stack. The default action is to use a **link** instruction only when a stack variable is referenced while the stack is temporarily displaced, or when the program is to be examined with a source-level debugger. Its use can be forced with the **-Of1** command line flag. When the **link** instruction is not used and a function parameter expects the top of the stack to be free a **subq.l #4, sp** instruction is inserted at the start of the function to free up the top of the stack.

The **movem** instruction (**move** or **movea** instruction if only one register is saved) saves on the stack the registers that must remain unmodified across a function

call. If the **link** instruction is used, the **movem** instruction moves the registers to be saved into stack space reserved by the **link**. If the **link** is not used, the **movem** instruction pushes the registers to be saved onto the stack. A second **movem** instruction restores the saved registers before the function returns.

When the **link** instruction is present, the function parameters are referenced relative to the frame pointer — register **a6**. The offset from register **a6** required to reference the leftmost function parameter (i.e., the last parameter pushed onto the stack) is eight bytes: four bytes for the function return address and four bytes introduced by the **link** instruction.

When the **link** instruction is not present, the function parameters are referenced relative to the stack pointer — register **a7** (also referred to as **sp**). The offset from register **a7** required to reference the leftmost function parameter is determined as follows:

- Four bytes for the function return address.
- Four bytes for each data or address register pushed onto the stack.
- Ten bytes for each floating-point register pushed onto the stack.
- Four bytes introduced by the **addq.l #4, sp** instruction, if present.

2.11.4. Returning Values

Functions that return scalar (nonaggregate) data types return their data in one of several registers. Integer data types are returned in register **d0**. Address pointers are returned in register **a0** (**d0** when the **-XA** flag is specified). Floating-point data types are returned in register **fp0**. The type of the return register is determined by the data type specified in the function declaration.

Functions that return a structure are significantly more complicated than functions that return a scalar data type. Inside the called function, the structure to be returned is copied into stack space that is allocated by the calling function. Information about the reserved stack space is passed to the called function as an address pointer that is pushed onto the stack immediately before the function call (as if it were the function's leftmost argument). Immediately before the function returns, the structure is copied into the reserved space. The address of the reserved space is returned in register **a0**. Immediately after the function returns, the pointer in **a0** is used to copy the returned structure to its destination.

The structure return mechanism can be further clarified using a pair of examples. Example A shows a function returning a structure, and Example B uses structure pointers to demonstrate (at the C level) what the compiler generates internally when a function returns a structure. These examples generate essentially identical code.

Example A

```
struct s {
       int a;
       int b;
};
struct s f();
struct s x;
main()
{
       x = f(1,2);
}

struct s f(int i,int j)
{
       struct z;
       z.a = i;
       z.b = j;
       return z;
}
```

Example B

```
struct s {
       int a;
       int b;
};
struct s *f();
struct s x;
main()
{
       struct s tmp;

       x = *f(&tmp,1,2);
}

struct *f(strct_ptr,i,j)
struct s *strct_ptr;
{
       struct z;

       z.a = i;
       z.b = j;
       *strct_ptr = z;
       return strct_ptr;
}
```

Warning: A function that returns anything other than an integer *must* be declared in a file before it is called. Failure to do so could result in unexpected behavior. Without a declaration in scope, a called function is assumed to return an integer; therefore, its return value is expected to be in **d0**. However, if the function actually returned a pointer, the return value would be found in **a0**. Similarly, if the function returned floating-point data, the return value would be in **fp0**. Even if the function returned an object of type **char** or **short int**, the upper portion of **d0** might contain invalid data that would result in a corrupted integer value.

Warning: A function that returns a structure *must* be declared in a file before it is called, even when the return value is not used. Without a declaration in scope, the pointer to the structure return area is not pushed onto the stack. The called function, which assumes that the memory pointer has been pushed, cannot access its parameters correctly and may corrupt memory when it attempts to return a structure.

2.11.5. Register Usage

Data, address, and floating-point registers are used to hold both the values of automatic variables and the intermediate results generated during the evaluation of an expression. The registers used to hold intermediate expression values are referred to as scratch registers. Registers **d0–d2**, **a0–a1**, and **fp0–fp1** are used as scratch registers by the compiler; their values are not guaranteed across function calls. When the **-X2** command line flag is specified, the value of register **d2** is guaranteed across function calls. Registers **d3–d7**, **a2–a5**, and **fp2–fp7** must remain stable across function calls and are thus required to be saved and restored in any function that uses them. Registers **a6** (frame pointer) and **a7** (stack pointer) must also remain stable across function calls. The **unlk** instruction restores registers **a6** and **a7**.

2.12. Compiler-Generated Function Calls

The compiler usually generates in-line assembly code when translating C programs. Sometimes, however, the compiler calls library functions to perform operations when in-line code would be inefficient or when special debugging options have been requested. The compiler-generated function calls fall into the following three categories:

1. calls to integer arithmetic functions
2. calls to floating-point software functions
3. calls to debugging functions

Compiler-generated function calls do not follow the standard C function calling conventions. For example, when calling internal integer arithmetic functions, parameters are passed in registers instead of on the stack. The following sections describe the special calling conventions used by each category of compiler-generated function calls.

2.12.1. Internal Integer Arithmetic Functions

The 68000 compiler, **com68**, generates function calls to perform integer arithmetic when in-line code would require an unacceptable amount of space. It also generates function calls to perform some division and modulus operations. The size and type of the operands determine whether a function call is generated. For example, the division of two signed 16-bit operands will be handled by in-line code, but the division of two signed 32-bit operands will be handled by a function call. Table 2.5, Integer Arithmetic Functions, lists the functions that are called by the 68000 compiler to perform the specified integer arithmetic operations.

Function Name	Operation
__du16u16	Divide 16-bit unsigned by 16-bit unsigned
__ds32s32	Divide 32-bit signed by 32-bit signed
__du32u32	Divide 32-bit unsigned by 32-bit unsigned
__ds16u16	Divide 16-bit signed by 16-bit unsigned
__ms16u16	Mod 16-bit signed with 16-bit unsigned
__mu16u16	Mod 16-bit unsigned with 16-bit unsigned
__ms32s32	Mod 32-bit signed with 32-bit signed
__mu32u32	Mod 32-bit unsigned with 32-bit unsigned

Table 2.5: Integer Arithmetic Functions

Arguments are passed to the functions listed in Table 2.5 in registers **d0** and **d1**; they are not passed on the stack. Results are always 32 bits and are returned in register **d1** (basically, **d1 = d1 / d0** or **d1 = d1 % d0**). The functions that operate on 16-bit operands use only registers **d0** and **d1**. The functions that operate on 32-bit operands also use registers **d2**, **a0**, and **a1**.

The following example, which shows both a 68000 C file and the generated assembly code, illustrates the generation of an internally generated function call:

```
extern int i, j;
void f( void )
{
       i = i / j;
}
```

```
move.l      _i,d1
move.l      _j,d0
jsr         __ds32s32
move.l      d1,_i
rts
```

2.12.2. Internal Floating-Point Functions

The Sierra C compiler supports two different floating-point formats, each one being automatically used in specific cases. The supported formats include the TI BCD floating-point format and an IEEE format which exists only in the compiler, not on the TI-89 / TI-92 Plus. Floating-point operations on constants are performed in the IEEE format by the compiler while all other floating-point operations are handled by internally generated calls to the BCD floating-point routines on the TI-89 / TI-92 Plus. Floating-point constants are converted to BCD if they are used as operands in the TI BCD floating-point routines. It is important to remember that in rare cases, it is possible that a value computed in the IEEE format and converted to BCD may differ from the result of the identical operation if performed by the TI floating-point routines due to the difference in accuracy between IEEE and BCD. Floating-point constant operations should be used with caution for this reason, although since the IEEE format is comparable to 20 BCD digits and the TI BCD values have 16 digits, differences will be extremely rare.

```
double flt1=1024. * 16.;         /* IEEE operations */
double flt2;
flt2 = flt1 * 4.;       /* TI BCD floating-point routines */
```

When generating code for the TI BCD floating-point routines, floating-point registers **fp0** through **fp7** correspond to stack frame locations (**-10, a6**) through (**-80, a6**), respectively; each location occupies ten bytes. To minimize the amount of code needed to make an internally generated call, the compiler goes through an interface function on the TI-89 / TI-92 Plus, __**bcd_math**. Calls to the floating-point interface function differ from calls generated for C language function calls in two major respects:

- Operand information, including the specification of both source and destination registers (when applicable), is encoded into a single two-byte argument.
- The called (not the calling) function restores the stack upon function return.

When using the function call to __**bcd_math**, a two-byte code word is inserted into the instruction sequence immediately following the function call. The code word fully describes the floating-point operation, the size of the operands, and the effective addresses of both the source and destination operands. If both the source and destination operands are registers (data and/or emulation floating-point registers), no information other than that supplied by the code word is required.

Figure 2.4 explains how to decipher information in the floating-point emulation code word. The operator and size components of the code word should be self-explanatory. Source and destination registers refer to the actual processor data registers and the emulation floating-point registers that correspond to locations on the stack frame (as explained above). If the destination operand is a register, the operator, size, and source operand determine whether it references a data register or a floating-point register.

The TI software emulation is patterned after the 68881/2 floating-point coprocessor (see section **3.1.2 Prerequisite Reading** for a list of references if more information is desired). The same rules and restrictions that apply to 68881/2 instruction operands apply to corresponding emulation instruction operands. If one of the operands is specified by its absolute address, the 32-bit address is pushed onto the stack immediately before the function call. If an operand is specified to be an immediate **short**, immediate **long**, or frame offset, the 16-bit or 32-bit immediate value or the 16-bit **a6**-relative frame displacement is inserted into the instruction sequence immediately following the code word. From the information in the code word, the called function adjusts its return address to skip over the code word and the operand (if present).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Operator				**Size**			**Source Operand**					**Dest. Operand**			

fcmp	0	byte	0	fp0	0	r0 (fp or d)	0
fadd	1	word	1	|	|	|	|
fdiv	2	long	2	fp7	7	r7 (fp or d)	7
fmul	3	single	3	d0	8	frame offset	8
fsub	4	double	4	|	|	effective address	9
fintrz	5	extend	5	d7	15	return register	10
fmove	6			immed. long	16		
fneg	7			immed. short	17		
ftst	8			frame offset	18		
fbcc	9			effective address	19		
				immed. zero	20		

Figure 2.4: Floating-Point Emulation Code Word

The following example demonstrates the interface to the TI BCD floating-point routines. Shown below is a sample C listing, followed by the code that is generated after compilation.

```
double area, radius;
void circle( void )
{

             area = 3.14159* radius * radius;

}

68000 C Compiler 3.2h Copyright 1987-99 by Sierra Systems. All rights
reserved.

             .opt     proc=68000
             .file    "test.c"
             .comm    _area,10,2
             .comm    _radius,10,2
                                ; 4 double area, radius;
                                ; 5 void circle( void )
             .text
             .align   2
             .globl   _circle
             .def  _circle\  .val  _circle\  .scl  2\  .type  0xd0020\
             .endef
_circle:
             link     a6,#-36
                                ; 6 {
                                ; 9 area = 3.14159* radius * radius;
;            fmove.d  _radius,fp0
             move.l   a0,-(sp)
             lea      _radius,a0
             move.l   (a0)+,-10(a6)
             move.l   (a0)+,-6(a6)
             move.w   (a0)+,-2(a6)
             movea.l  (sp)+,a0
;            fmul.d   _radius,fp0
             pea      _radius
             jsr      __bcd_math
             .short   0x3930
;            fmul.d   #3.14159000000000000000e+00,fp0
             jsr      __bcd_math
             .short   0x3900
             .short   0x4000
             .long    0x31415900
             .long    0x0
;            fmove.d  fp0,_area
             lea      _area,a0
             move.l   -10(a6),(a0)+
             move.l   -6(a6),(a0)+
             move.w   -2(a6),(a0)+
                                ; 12 }
             unlk     a6
             rts
             .def  _circle\  .val  .\  .scl  -1\
             .endef
```

2.12.3. Debugging Functions

The compiler can be directed to insert calls to debugging functions in the generated assembly code using the **-XS**, **-Xs**, and **-Xc** command line flags. The **-XS** and **-Xs** flags cause the compiler to place calls to **__stk_ck** (i.e., **_stk_ck** defined as a C function) at the beginning of each function. The **-Xc** flag causes the compiler to place a call to **__line_ck** (i.e., **_line_ck** defined as a C function) following each line of C source code.

Both the **_stk_ck** and **_line_ck** functions are intended to be provided by the user. These functions provide a mechanism to check various aspects of a program. Although the **_stk_ck** function will typically compare the current stack pointer offset and the amount of stack space needed by the calling function to the total amount of available stack space, on the TI-89 / TI-92 Plus, this is done through hardware. Even though the traditional use for **_stk_ck** is no longer necessary, the function may be used for any other debugging purpose desired. The **_line_ck** function could be used to locate the position in a program where a particular memory location is corrupted.

The **-XS** flag causes the compiler to insert calls to **_stk_ck** at the beginning of every function in the file. The amount of stack space used by the function is available to **_stk_ck** in register **d0**. If the **_stk_ck** function uses any nonscratch registers (i.e., **d3–d7**, **a2–a7**, or **fp2–fp7**), it must save and restore them.

The **-Xs***size* flag is identical to the **-XS** flag with the exception that it causes the compiler to insert the call to **_stk_ck** only when a function is expected to take more than *size* number of bytes of stack space. If no value is specified, the default value of 40 bytes is used.

The **-Xc** flag causes the compiler to insert a call to **_line_ck** following each line of compiled C source code. No information is passed to **_line_ck**. The **_line_ck** function can be a very useful debugging aid, and its uses are left up to your imagination. Unlike other functions, **_line_ck** must save and restore all registers it uses including scratch registers **d0–d2**, **a0–a1**, and **fp0–fp1**. In addition, **_line_ck** must save and restore the condition codes in the status register, because the function call may occur between the time a condition code is set and when it is used.

> **Note:** The compiler adds a leading underscore (_) to all function names and external variables (unless the **-Xu** flag is specified). For example, the C language functions *abc*( ) and *_xyz*( ) will appear in an assembly language listing as *_abc* and *_ _xyz*, respectively.

2.13. Sections

The TI-89 / TI-92 Plus SDK includes example invocations of the compiler. You must use the sections as shown in those files when compiling applications. The compiler places information into four different sections for TI-89 / TI-92 Plus applications: **.text**, **.data**, **.const**, and **.bss** (blank static storage). Initialized static data that has not been declared **const** is placed into **.data**. Uninitialized static data is placed into **.bss**. The **.bss** section is set to zero during app initialization. For more detailed information on application data storage and initialization, see chapter **7. Flash Application Layout** in the TI-89 / TI-92 Plus Developers Guide supplied with the TI-89 / TI-92 Plus SDK.

2.14. Static Storage Initialization

A data object with static storage duration can be assigned an initial value when it is declared. The initializing value is placed directly into the **.text**, **.const**, or **.data** section of the program by the compiler; the assignment is not made at run-time. If a static data object is not initialized in a declaration, the data object is assigned to the **.bss** section where it is set to zero at the start of program execution.

An initializer for an integer or floating-point data object must be an expression that evaluates to a constant during compilation. A compile-time constant expression may contain an arbitrary number of operators and subexpressions and include both integer and floating-point constants. The same data type conversions that apply across an assignment at run-time also apply to compile-time initializations. The following are examples of legal compile-time initializations:

```
int a = 50;
double b = 3.1415926;
int c = 2 * (5 + 7 / 3) - (122 && 15);
int *d = &c + 50;
char *e = (char *)0x80000;
```

When a scalar (pointer or arithmetic object) is initialized, a single expression optionally enclosed in a single set of braces, is permitted in the declaration.

See section **2.12.2 Internal Floating-Point Functions** for more information on initializing floating-point data objects. An initializer for a pointer must be an expression that evaluates to an integer constant at compile-time or the address of a static object, plus or minus a constant expression. When the address of an object appears in an initializer the object must have been declared previously in the same file. When a nonzero constant expression is used to initialize a pointer, the expression must be cast to the appropriate pointer type.

When the initialized variable is an aggregate (structure or array), the initializer consists of a brace-enclosed list of comma-separated initializers. The initializers in the list are applied to the aggregate in increasing member or subscript order. If the aggregate contains members that are also aggregates, the same rules apply recursively to the subaggregates. If there are fewer initializers in a brace-enclosed list than there are members in the aggregate, the remaining members are initialized to the value zero. It is illegal for there to be more initializers in a brace-enclosed list than there are members in the aggregate. For initialization purposes a union is treated as a structure that contains a single member, where the single member is the first member of the union.

The following is a fully initialized array of structures:

```
struct {
       int a;
       int b[3];
} c[2] = { {1, {2,3,4}}, {5, {6,7,8}} };
```

The initial values for the array c are as follows:

```
c[0].a = 1, c[0].b[0] = 2, c[0].b[1] = 3, c[0].b[2] = 4
c[1].a = 5, c[1].b[0] = 6, c[1].b[1] = 7, c[1].b[2] = 8
```

The following is a partially initialized array of structures:

```
struct {
        int a;
        int b[3];
} d[2] = { {1, {2}}, {5, {6,7}} };
```

The initial values for the array d are as follows:

```
d[0].a = 1, d[0].b[0] = 2, d[0].b[1] = 0, d[0].b[2] = 0
d[1].a = 5, d[1].b[0] = 6, d[1].b[1] = 7, d[1].b[2] = 0
```

Note: When an aggregate is initialized, its scalar members can also be brace enclosed. The following two declarations are equivalent:

```
int a[3] = { 10, 20, 30 };
int a[3] = { {10}, {20}, {30} };
```

The scalar-level braces are rarely used, however.

The braces surrounding subaggregate initializers are not always necessary. When a subaggregate initializer does not begin with a brace, only enough initializers are taken from the list to account for the members of the subaggregate. If there are too few initializers in the list to initialize the subaggregate, the remainder of the subaggregate is padded with zeros. If there are more initializers in the list than needed to initialize the subaggregate, the remainder of the initializers in the list initialize the next member of the aggregate of which the subaggregate is a member. For example, the following two declarations are equivalent:

```
short x[2][3] = { {1, 2, 3}, {4, 5, 6} };
short x[2][3] = {1, 2, 3, 4, 5, 6};
```

The following two examples, however, are not equivalent:

```
short x[2][3] = {1, 2};
short x[2][3] = {{1}, 2};
```

The first defines an array in which x[0][0] = 1, x[0][1] = 2, and all other elements are zero; the second defines in array in which x[0][0] = 1, x[1][0] = 2, and all other elements are zero. The following declaration is illegal:

```
short x[2][3] = {1, {2,3}};
```

The declaration initialization list can also be used to establish the size of an array. In the following example, the array is determined to have four elements:

```
double z[] = {1.2, 2.3, 4.555, 3.14};
```

Finally, a character array may be initialized by a character string, optionally enclosed in braces. Successive characters from the character string — including the terminating null character if there is room or if no size has been specified for the array — are used to initialize the array. The following example illustrates equivalent methods for declaring arrays s and t:

```
char s[3] = "abc";
char s[] = {'a', 'b', 'c'};
char s[] = {"a"
            "b"
            "c"
           };

char t[] = "xyz";
char t[] = {'x', 'y', 'z', '\0'};
```

The following two declarations are similar, but not identical:

```
char *p = "abc";
char q[] = "abc";
```

Both p and q can be used to reference the characters ' a ', ' b ', and ' c '. However, only q can be used to modify the character values. q is an array, the contents of which are modifiable. p is a pointer initialized to point to a character string, an array of characters which is not modifiable.

2.15. Compiler Algorithms

This section describes the register allocation algorithm and the criteria used to select between the three different **switch** statement algorithms.

2.15.1. Register Allocation

The allocation of automatic variables to machine registers (data, address, and floating-point) is a two-phase process. First, register candidates are selected. Within a function there can be up to 32 register candidates. The default behavior of the compiler is to select first the automatic variables that are declared with the **register** keyword. If fewer than 32 automatic variables are declared **register**, additional automatic variables that qualify for register allocation are classified as register candidates. An automatic variable qualifies for allocation to a register if its address is not taken and it is not an array or structure type. If there are 32 or fewer automatic variables in a function that qualify for placement into registers, they are all classified as register candidates. If there are more than 32 variables that qualify, the qualifying variables that are not already declared **register** are prioritized based on the number of times they appear in the function. On the basis of their assigned priorities, the remaining variables are classified as register candidates until all 32 slots are filled.

The **`-Or#`** flag can be used to influence which variables are selected to be register candidates. The register candidate selection algorithm described above is the default algorithm specified by the **`-Or2`** flag. The **`-Or0`** flag specifies that no variables are to be considered register candidates. The **`-Or1`** flag specifies that only variables declared **register** are register candidates. The **`-Or3`** flag forces the **register** declaration to be ignored, but otherwise follows the default selection algorithm. The **`-Or3`** flag allows the compiler to use its own judgment to select the most appropriate register candidates. The **`-Or4`** flag prevents floating-point variables that are not declared **register** from being considered as register candidates. On an individual basis, automatic variables can be removed from consideration as register candidates by applying the address-of operator (`&`). For example, the C statement '`&count;`' will not generate any code and will guarantee that the variable **count** will not be placed in a register.

In the second phase of the register allocation process, the one or more lifetimes of each register candidate are identified. For example, if a variable is used as a loop counter in three non-nested loops and the variable is not referenced outside any of the loops, then the variable is considered to have three discrete lifetimes.

Each variable lifetime is assigned a weight that is determined by the number of times the variable is referenced, how deeply nested (inside loops) the references are, and how many instructions the lifetime spans. The greater the number of times the variable is referenced and the more deeply nested the references are, the higher the assigned priority; the more instructions the lifetime spans, the lower the assigned priority. The lifetimes are then sorted by priority in descending order. Finally, attempts are made to fit as many variable lifetimes into as few machine registers and stack locations as possible. Going through the prioritized lifetimes list, as many nonoverlapping lifetimes as possible are assigned to the first register. After going through the entire prioritized list, the allocation process moves to the next register or stack location. Register and stack space allocation occur in the following order:

1. Fit integer type objects into data registers until there are no more integer type objects to be assigned or there are no more data registers to be allocated.
2. Fit pointer type objects into address registers until there are no more pointer type objects to be assigned or there are no more address registers to be allocated.
3. Fit integer type objects into address registers until there are no more integer type objects to be assigned or there are no more address registers to be allocated.
4. Fit floating-point type objects into floating-point registers until there are no more floating-point type objects to be assigned or there are no more floating-point registers to be allocated.
5. Fit ten-byte objects into ten-byte units on the stack until there are no more ten-byte objects to be assigned memory locations.
6. Fit eight-byte objects into eight-byte units on the stack until there are no more eight-byte objects to be assigned memory locations.
7. Fit four-byte objects into four-byte units on the stack until there are no more four-byte objects to be assigned memory locations.
8. Fit two-byte objects into two-byte units on the stack until there are no more two-byte objects to be assigned memory locations.
9. Fit one-byte objects into one-byte units on the stack until there are no more one-byte objects to be assigned memory locations.

The register allocation algorithm described above is extremely powerful and it has some interesting properties. A variable declared **register** may not make it into a register if all of its lifetimes are assigned low priorities during the second phase of the allocation process, whereas a variable not declared **register** may make it into a register if its lifetimes are assigned high priorities. Every variable in

a function can be assigned to the same register if they all have nonoverlapping lifetimes. Different lifetimes of the same variable can be allocated to different registers or a combination of registers and stack locations.

2.15.2. Switch Statements

The compiler uses three different code generation algorithms to implement the C language **switch** statement. The algorithm employed for a given **switch** statement depends on the number of **case** statements and the range of **case** statement values.

If there are four or fewer case labels, the compiler generates a sequence of explicit tests against the case values. This is equivalent to the code generated by a series of **if-else** statements.

If there are more than four case labels, the **switch** algorithm selected is determined by both the number of **case** statements and their values. If the case value density is sufficiently high, a jump table is generated; otherwise, the compiler generates code that performs an in-line binary search to locate the appropriate **case** statement.

A jump table is generated when at least one third of the possible integral values over the case value range are represented by **case** statements (i.e., when the difference between the highest and lowest case values divided by the total number of **case** statements is less than three). For example, a jump table would not be generated for the following **switch** statement:

```
int i;
f()
{
       switch(i) {
       case 4:  ... ;
       case 8:  ... ;
       case 12: ... ;
       case 16: ... ;
       case 20: ... ;
       default: ... ;
       }
}
```

In this example, the binary search algorithm is used because the case density is not high enough to justify a jump table. The difference between the maximum and minimum **case** values divided by the number of **case** statements is not less than three (i.e., (20 - 4) / 5 > 3).

Typically, the binary search algorithm is more compact than the jump table, and the jump table algorithm is faster than the binary search. When speed is more important than size, dummy case statements can be added to the **switch** statement in order to force the use of a jump table.

By default, jump tables use a word offset to specify the location of the code associated with a particular case label. The word table handles most **switch** statements likely to be encountered. In the unlikely event that the distance between the jump table and the code associated with a case label is greater than 32767, the **`-Xl`** (X el) command line flag can be specified to force the jump table to use long offsets.

2.16. The C Preprocessor

The C preprocessor is a simple but powerful macro processor that manipulates the text of a C source program before it is passed onto the compiler proper. The preprocessor is typically used to simplify both the writing and maintenance of a C source program. Controlled by preprocessor directives embedded in the source file, the preprocessor can do the following:

1. Insert the contents of other files in the source.
2. Conditionally suppress portions of the source.
3. Make macro substitutions in the text.

The directives and operators recognized by the preprocessor are as follows:

#define	**#endif**	**#include**	**defined**
#elif	**#if**	**#line**	**#**
#error	**#ifdef**	**#pragma**	**##**
#else	**#ifndef**	**#undef**	

All preprocessor directives begin with a pound sign (#) and occupy a single source line. The ' # ' character is permitted to be preceded and/or followed by whitespace (i.e., tabs, spaces, and comments). A single source line can be extended to include multiple lines in the file by inserting a backslash (\ ) before all but the last newline.

Character sequences recognized as tokens by the preprocessor are: header names within a **#include** directive, identifiers, constants, string literals, punctuation, and non-whitespace characters that are not one of the previously mentioned.

2.16.1. Source File Inclusion

The **#include** directive causes the contents of the named file to be treated as if it had appeared in place of the directive itself. The included file can contain anything that is permitted in the including file, including other **#include** directives. The argument to the **#include** directive, used to specify the file for inclusion, can take one of three forms.

A preprocessing directive of the following form specifies that the filename (optionally preceded by directory information) is to be searched for in a series of standard directories:

```
#include <filename>
```

The standard directories can be defined both on the compiler command line using the **-I** flag and by setting the **INCLUDE68** environment variable. The filename is searched for until it is located, first in the directories named on the command line (in the order listed) and then in the directory(ies) defined by **INCLUDE68**. If **INCLUDE68** is not set, the **include** subdirectory of the directory specified by the **SIERRA** environment variable is searched.

A preprocessing directive of the following form specifies that the filename (optionally preceded by directory information) is to be searched for in association with the original source file:

```
#include "filename"
```

Alternatively, the filename is searched for in association with the including file when the **-R** compiler command line flag is specified. If the filename is not located in association with the appropriate file, the **#include** directive is processed as if the filename had been enclosed in angle brackets as in the previously described form.

A preprocessing directive of the following form is also permitted:

```
#include preprocessor_tokens
```

Macro names among the preprocessor tokens after the **#include** directive are processed the same as any other macro names in the file. The directive that results after macro substitution must match one of the two previous forms and is processed accordingly.

2.16.2. Conditional Compilation

The **#if** directives (i.e., **#if**, **#else**, **#elif**, **#ifdef**, **#ifndef**, and **#endif**) allow selected lines of text in the file to be conditionally included or excluded from further processing.

Each **#if** directive in a file must be paired with a closing **#endif** directive. Zero or more **#elif** (similar to **#else** followed by **#if**) directives can appear between a **#if** and **#endif** pair and at most one **#else** directive can appear between the pair. If the **#else** directive does appear, it must be the last directive before the **#endif**. These directives can be nested inside other **#if** directives.

In the following **#if** directive sequence, *text block 1* remains in the file if the constant expression following the **#if** evaluates to a nonzero value, *text block 2* remains in if the first constant expression evaluates to zero and the constant expression following the **#elif** evaluates to nonzero, and *text block 3* remains in if both constant expressions evaluate to zero:

```
#if constant_expression
       text block 1
#elif constant_expression
       text block 2
#else
       text block 3
#endif
```

A constant expression that follows a **#if** or **#elif** must contain only integral constants (including character constants that may contain escape sequences), and it must not contain the **sizeof** operator, a cast operator, or an enumeration constant. However, it may contain unary expressions of either of the following forms:

```
defined identifier

defined (identifier)
```

These evaluate to 1 if the identifier is currently defined as a macro and to 0 otherwise. All identifiers in the constant expression currently defined as macros (except those that are arguments to the defined operator) are replaced and identifiers not currently defined are replaced by the constant 0. After all the above substitutions are performed, the constant expression is evaluated (treating all constants as 32-bit integers) following the same rules used to evaluate any other C expression.

Preprocessing directives of the following forms selectively include or exclude the block of text that follows based on whether the identifier is or is not currently defined as a macro name:

```
#ifdef identifier

#ifndef identifier
```

The above directives are functionally identical to the following:

```
#if defined (identifier)

#if !defined (identifier)
```

2.16.3. Macro Replacement

The **#define** directive is used to define a macro. It has two forms, depending on whether the identifier to be defined is immediately followed by a left parenthesis.

A preprocessing directive of the following form defines an object-like macro:

```
#define macro_name replacement_list
```

It causes each subsequent instance of the macro name to be replaced by the replacement list that constitutes the remainder of the directive. The replacement list is then rescanned for more macro names as specified below.

A preprocessing directive of the following form defines a function-like macro with arguments that is similar in appearance to a function call:

```
#define macro_name(identifier_list) replacement_list
```

The parameters are specified by the optional identifier list. Each subsequent instance of the function-like macro name followed by a left parenthesis introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition. (A function-like macro name that is not followed by a left parenthesis is not recognized as a macro name and no replacement is made.) The replaced sequence is terminated by the matching right parenthesis, skipping intervening matched pairs of left and right parenthesis. Within the sequence that comprises an invocation of a function-like macro, a newline is considered a normal whitespace character.

The sequence of tokens bounded by the outermost matching parentheses forms the argument list for the function-like macro. The individual arguments within the list are separated by commas; commas in the list enclosed within nested parenthesis do not separate arguments. The number of comma-separated arguments in the list must match the number of parameters in the function-like macro definition.

2.16.3.1. Argument Substitution

After the arguments in a function-like macro are identified, argument substitution takes place. Each parameter in the replacement list, unless preceded by a ' # ' or ' ## ' preprocessor operator or followed by a ' ## ' operator, is replaced by the macro expansion of the corresponding argument from the argument list. In other words, each argument, unless its parameter is associated with a ' # ' or ' ## ' operator, is fully expanded (i.e., all macros replaced) before it is substituted into the replacement list.

2.16.3.2. The # Operator (stringizing)

If a parameter in the replacement list is immediately preceded by the ' # ' preprocessing operator, the ' # ' operator and the parameter are both replaced by a string literal that contains the spelling of the corresponding argument. The argument (not expanded) is placed within a pair of double quotes with any whitespace before and/or after the argument removed. Other than the removal of leading and trailing whitespace, the exact spelling of the argument is retained in the string.

2.16.3.3. The ## Operator (concatenation)

If a parameter in the replacement list is immediately preceded or followed by the ' ## ' preprocessing operator, the parameter is replaced by the macro-expanded corresponding argument. For both object-like and function-like macro invocations, each instance of the ' ## ' preprocessing operator in the replacement list (not brought in by an argument) is deleted and the preceding and following preprocessing tokens are concatenated before the list is reexamined for macro names to be replaced. The token created by concatenation is then available for further macro replacement as is any other token in the replacement list.

2.16.3.4. Rescanning and Further Replacement

After all parameters in the replacement list have been substituted as described above, the resulting tokens are rescanned for more macro names to replace. If the name of the macro being replaced is encountered during this scan of the replacement list, it is not replaced. Further, if any nested replacements encounter the name of the nested macro being replaced, it is not replaced. These nonreplaced macro names are not available for further replacement even if they are examined again in a context where they would otherwise be replaced. Finally, if the fully expanded preprocessor token sequence resembles a preprocessing directive, it is not recognized as such.

2.16.4. Macro Redefinition

An identifier currently defined as an object-like macro may be redefined by another **#define** directive provided that the second definition is also an object-like macro definition and the two replacement lists are identical. An identifier currently defined as a function-like macro may be redefined provided that the second definition is also a function-like macro definition and both the number of parameters and the replacement lists are identical.

Two replacement lists are considered identical if and only if the two lists have identical spelling and whitespace separation. If it is necessary to redefine a macro and the second definition has a different replacement list, a **#undef** directive must appear before the redefinition.

2.16.5. Macro Examples

The following example illustrates the rules for redefinition, tokenization, macro replacement, and replacement list reexamination:

```
#define x           3
#define q           x
#define f(i)        f(q + (i->m))
#define b           b + c
#undef x
#define x           5
#define z(a)        a
#define g           f
#define j           )
#define d           +
#define e           +
#define r(a, b)     (a)(b)
#define s(a, b)     a(b)
#define y           g(y)


r(f, g)
s(f, q)
g(36j
#undef q
#define q           d
f(b + c)
f(y)
35d+12
35e+12


/* Results After Preprocessing */

(f)(5)
f(5 + (4->m))
f(36)
f(+ + (b + c + c->m))
f(+ + (f(y)->m))
35++12
35e+12
```

The following example illustrates the rules for concatenating tokens:

```
#define paste(a,b)          a ## b
#define opnd_info(n)        op##n##_info(size##n(),type##n())
#define HITHERE             hello, world!

paste(HI,THERE)
opnd_info(2)

/* Results After Preprocessing */

hello, world!
op2_info(size2(),type2())
```

The following example illustrates the rules for creating string literals:

```
#define stringize(s)    # s
#define expand(s)       stringize(s)
#define max(a,b)        ((a) > (b) ? (a) : (b))
#define show_macro(m)   printf(#m " becomes " expand(m)"\n")

show_macro(max(a,b))

/* Results After Preprocessing */

printf("max(a,b)" " becomes " "((a) > (b) ? (a) : (b))""\n")
```

After string concatenation, the above result appears as follows:

```
printf("max(a,b) becomes ((a) > (b) ? (a) : (b))\n")
```

Space around the ' # ' and ' ## ' operators is optional.

2.16.6. Line and Name Control

A preprocessing directive of the following form causes the compiler to behave as if the line number of the next source line is the number specified by the decimal constant in the directive:

```
#line decimal_constant
```

A preprocessing directive of the following form sets the line number as specified above and changes the presumed name of the source file to be the filename shown in the string literal:

```
#line decimal_constant "file_name"
```

Macro names among the preprocessor tokens following the **#line** directive are processed the same as any other macro names in the file. The directive that results after macro substitution must match one of the two previous forms and is processed accordingly.

2.16.7. Error Directive

A preprocessing directive of the following form causes the compiler to produce an error message that includes the macro expansion of the specified sequence of preprocessing tokens:

```
#error preprocessing_tokens
```

For example, the following sequence causes generation of the error message shown below if SIZE is greater than 1024:

```
#if SIZE > 1024
#line 150 "test.c"
#error SIZE is greater than 1024!
#endif

test.c, 150: SIZE is greater than 1024!
```

2.16.8. Pragma Directive

A preprocessing directive of the following form is a pragma:

```
#pragma preprocessing_tokens
```

A pragma causes the compiler to behave in specified ways. The functionality of a **#pragma** directive is similar to that of a command line flag, except that the specified behavior can be embedded in the source file and turned on and off multiple times during a single compilation. If the directive following the **#pragma** is not recognized, the pragma directive is ignored. Refer to section **2.4 Pragma Directives**, for a description of the pragmas supported by Sierra C.

2.16.9. Trigraph Sequences

A trigraph sequence is a sequence of three characters (??*x*) that maps into a single character. They are used to facilitate the writing of C programs on terminals that do not support all the characters required by the C language. The trigraph conversion capabilities are enabled by specifying the **-T** flag on the compiler command line. The supported trigraph sequences are as follows:

```
??=      ⇒     #
??(      ⇒     [
??/      ⇒     \
??)      ⇒     ]
??'      ⇒     ^
??<      ⇒     {
??!      ⇒     |
??>      ⇒     }
??-      ⇒     ~
```

For example, the following two source lines are equivalent:

```
int a??(10??) = ??<1, 2, 3??>;

int a[10] = {1, 2, 3};
```

2.16.10. Comment Delimiters

In addition to the standard ANSI C comment delimiters slash-asterisk (/*) and asterisk-slash (*/), the C++ style comment delimiter slash-slash (//) is supported by the compiler. Everything from the // to the end of the current line is recognized as a comment. Note, however, that the use of the // comment delimiter is not recommended since the resulting code will be nonportable.

2.16.11. Predefined Macro Names

The following names are predefined by the compiler:

__DATE__	The date that the source file was compiled given as a string literal of the form "*Mmm dd yyyy*", where the first character of *dd* is a space if the value is less than 10.
__FILE__	The presumed name of the source file given as a string literal (e.g., "file_name").
__FLOAT__	Not supported by Texas Instruments, however, **__FLOAT__** is recognized as a reserved name by the compiler.
__INT__	A macro that expands to 16 or 32 to indicate whether the compiler is interpreting objects of type **int** as 16 or 32 bits. The macro expands to 16 only when the **-XI** command line flag is specified.
__LINE__	The line number of the current source line given as a decimal constant.
__PCREL__	A macro that expands to 0 if position-independent code is not being generated and 1 if position-independent code is being generated, as specified by the **-Xp** command line flag.
__TIME__	The time that the source file was compiled given as a string literal of the form "*hh:mm:ss*".
__SIERRA__	A macro that expands to identify the version number of the compiler.

These macro names can be neither defined nor undefined with the **#define** or **#undef** preprocessing directives.

The following example illustrates the use of predefined macros:

```
printf(__FILE__": compiled on "__DATE__" at "__TIME__"\n");
/* Results After Preprocessing and String Concatenation */
printf("test.c: compiled on Aug 29 1992 at 17:35:23\n");
```

2.17. Compiler Error Messages

All error messages are listed alphabetically. Warning messages are alphabetized under *w* for warning. When searching for a message, ignore all identifier names, filenames, and flags (shown in italics here). Also, ignore the hyphen character and the leading word *the*. Uppercase and lowercase letters are treated the same.

`-flag argument too long`

The argument to a **-D** or **-U** flag exceeds 512 characters.

`array of functions is illegal`

Functions cannot be elements of an array. An array of pointers to functions is legal.

`array size is unknown`

An automatic or uninitialized static array is declared with an empty dimension.

`asm keyword requires a character string`

The argument specified inside the parentheses following the **asm** keyword must be a character string enclosed in double quotes.

`bad call to system function identifier()`

A call to the system function *identifier*( ) failed (returned with an error status).

`bit field illegal outside of structure`

Bit fields are permitted only in structures (not unions).

`bit field size is zero`

The size of a named bit field must be greater than zero.

`bit field size too large`

Depending on the type of the bit field, the length of a bit field cannot exceed the number of bits in a **char** (8), **short int** (16), **int** (16), or **long int** (32).

`cannot include file file`

The compiler could not open the given **#include** file.

`cannot open file for reading`

The given *file* could not be opened for reading.

`cannot open file for writing`

The given *file* could not be opened for writing.

`cannot re-open file`

The file *file*, which has been temporarily closed to permit the inclusion of deeply nested **#include** files, cannot be re-opened. In a multi-tasking system, this can occur when the *file* is removed by another task during compilation.

`case outside of switch`

A case label must appear inside a **switch** statement.

`case requires constant integer expression`

A case expression must evaluate to an integer constant.

`character constant too long`

A character constant contains more than four characters.

`character string too long`

A character string contains more than 512 characters. When adjacent character strings are concatenated, the 512-character limit applies to the resulting string.

`command line: -flag followed by invalid argument: string`

The given *string* is an invalid argument to the given *flag*.

`command line: -flag is an invalid flag`

The given *flag* is not recognized.

`command line: -flag requires an argument`

The given *flag* must be followed by an argument.

`command line: -i flag is illegal in a command file`

Command line include files cannot be nested inside each other.

`command line: more than two files are specified`

The compiler accepts at most two filenames on the command line (not counting filenames that are arguments to flags): an input file and an output file.

`compiler cannot recover from earlier errors`

The compiler must exit because previous errors are too serious to permit recovery.

`compiler error` *`number`* `(internal)`

An internal error has occurred in the compiler. Please submit a problem report on the TI web site.

`compiler error: move function call outside expression`

An internal error has occurred in the compiler. This problem will rarely occur, but when it does, use a temporary variable to move the function call outside the expression.

`constant integer expression required`

A constant integer expression is required. This error may occur when a floating-point or nonconstant expression is used in an array declaration, bit field specification, or **enum** definition.

`declaration for` *`identifier`* `is too complex`

The complexity of the declaration for the given *identifier* has exceeded an internal compiler limit.

`declaration is too complex`

The complexity of a declaration has exceeded an internal compiler limit.

`default outside of switch`

A default label must appear inside a **switch** statement.

`# directive syntax error`

A pound sign (#) is followed by an unrecognized directive.

`duplicate case` *`number`* `inside switch`

The given case *number* appears more than once inside a **switch** statement.

`duplicate default inside switch`

The default label appears more than once inside a **switch** statement.

`#elif following #else at same level`

A **#elif** directive cannot follow a **#else** directive without an intervening **#if**, **#ifdef**, **#ifndef**, or **#endif** directive.

`#elif without matching #if, #ifdef or #ifndef`

A **#elif** directive cannot appear without being preceded by a matching **#if**, **#ifdef**, or **#ifndef** directive.

`#else following #else at same level`

Once a **#if**, **#ifdef** or **#ifndef** directive has been matched by a **#else** directive, it cannot be matched by a **#elif** directive.

`#else without matching #if, #ifdef or #ifndef`

A **#else** directive cannot appear without being preceded by a matching **#if**, **#ifdef**, or **#ifndef** directive.

`#endif without matching #if, #ifdef or #ifndef`

A **#endif** directive cannot appear without being preceded by a matching **#if**, **#ifdef**, or **#ifndef** directive.

`empty character constant`

The empty character constant (i.e., two adjacent single quotes) is illegal.

`EOF encountered in definition of macro` *`identifier`*

An unexpected **EOF** (end of file) appeared in the definition of macro *identifier*.

`EOF encountered in filename`

An unexpected **EOF** (end of file) appeared in the filename of a **#include** directive.

`expression is too complex`

The complexity of an expression has exceeded an internal compiler limit.

`expression stack overflow: use option '-XE60'`

The compilers expression stack ran out of space. The compiler option **-XE#** allows you to increase the size of the expression stack to avoid this problem. The default size for the expression stack is 30.

`extern data declaration cannot be initialized`

A data declaration that includes the **extern** storage class specifier does not allocate memory; thus, memory cannot be initialized.

`filename too long`

A filename (including the full path) in a **#include** directive exceeds the limits of the host operating system. The limit may be exceeded when a **#include** path is appended to a standard directory search path.

`float illegal in switch`

The expression following the **switch** keyword must evaluate to an integer type.

`floating point divide by zero`

A compile-time floating-point division by zero occurred.

`floating point operand error`

A compile-time operand error has occurred.

`formal parameter declaration list illegal with prototypes`

A prototyped function definition cannot have a parameter list between the function's closing parenthesis and opening brace. A parameter declaration list in that position is legal only with non-prototyped function definitions.

`function` *`identifier`*`() is illegal in struct/union`

A function cannot appear in a structure or union.

`function calls are nested too deep`

Function calls are nested deeper than 20 levels.

`function cannot be initialized`

It is illegal to initialize a function.

`function cannot return a function`

It is illegal to declare a function as returning a function.

`function cannot return an array`

It is illegal to declare a function as returning an array.

`function declarations must be empty or include prototypes`

The parenthesis following the function name in a nondefinition function declaration must either contain prototype information or be empty.

`identifier missing from formal parameter prototype`

A type specifier appeared without an identifier in the prototype list of a function definition.

`IEEE nonaware comparison involving a NAN (not-a-number)`

If the **`-Xi`** flag was not used, an internal compiler error has occurred. Please submit a problem report on TI's web site.

`#if's are nested too deep`

The **#if**, **#ifdef**, and **#ifndef** directives cannot be nested deeper than 16 levels.

`#include's are nested too deep`

Included files cannot be nested deeper than 50 levels.

`generated code contains too many labels`

The assembly code generated by the compiler contained more than 50,000 compiler generated labels.

`illegal bit field size`

The size of a bit field must be greater than or equal to zero (greater than zero if it is a named bit field).

`illegal bit field type`

A bit field must be an integer type (**char**, **short int**, **int**, or **long int**).

`illegal break`

A **break** statement may appear only inside a **do**, **for**, **while**, or **switch** statement.

`illegal character` *`character`* `in #if or #elif expression`

The given *character* is illegal in the expression following a **#if** or **#elif** preprocessor directive.

`illegal character (`*`number`* `hex) in #if or #elif expression`

The nonprinting character with the given hexadecimal representation is illegal in the expression following a **#if** or **#elif** preprocessor directive.

`illegal character (0 hex) in macro definition`

A macro definition contains a null character.

`illegal character after macro name following -D flag`

The only non-whitespace character allowed after a macro name following a **`-D`** flag is ' = '. The ' = ' is then followed by the text of the macro definition.

`illegal character in octal constant`

An octal constant can contain only the digits 0, 1, 2, 3, 4, 5, 6, and 7.

`illegal combination of different structures, op =`

A structure cannot be assigned to a structure of a different type.

`illegal combination of different structures, op RETURN`

A function returning a structure must return a structure type that matches the function return type.

`illegal continue`

A **continue** statement may appear only inside a **do**, **for**, or **while** statement.

`illegal declaration`

Either a syntax error appeared in a declaration or a **typedef** was used in a declaration with other type specifiers.

`illegal empty dimension`

Only the first dimension of a multi-dimensional array may be left empty in a declaration.

`illegal extern/static initializer`

Data declared with static storage duration must be initialized with a constant expression or the address of an object or function (when the address is known by the time the program has been linked).

`illegal floating-point constant`

The floating-point constant contains a syntax error.

`illegal function`

An expression with type other than function type was used where an expression with function type was expected.

`illegal function identifier()`

The given *identifier* with type other than function type was used where an undefined identifier or identifier with function type was expected.

`illegal #ifdef or #ifndef argument`

The argument to a **#ifdef** or **#ifndef** directive is not a valid C identifier.

`illegal indirection`

Indirection using the indirection operator (*) or subscript operator ([ ]) was applied to a nonpointer object.

`illegal indirection on identifier`

Indirection using the indirection operator (*) or subscript operator ([ ]) was applied to the specified nonpointer object.

`illegal initialization: type = type`

An incompatible *type* was used to initialize a variable of type *type*.

`illegal initialization: bit field = type`

The constant used to initialize a bit field was not an integer type.

`illegal initialization: { nested too deep }`

The braces in an aggregate initialization are nested more deeply than the object being initialized.

`illegal initialization: { string nested too deep }`

The braces surrounding the string initializer of a character array are nested more deeply than the objects being initialized.

`illegal left hand side of assignment`

The left operand of an assignment must be a modifiable value.

`illegal left hand side of assignment, op operator`

The left operand of the given assignment *operator* must be a modifiable value.

`illegal operation on void type, op CAST`

It is illegal to cast a **void** type to any other type.

`illegal pointer subtraction`

It is illegal to subtract pointers of different types.

`illegal redefinition of macro` *`identifier`*

The given *identifier* is already defined as a macro and the new macro replacement list does not match exactly the already existing macro replacement list.

`illegal storage class`

The specified storage class is illegal in the present context. For example, the **auto** and **register** storage class specifiers cannot be used outside a function or in conjunction with the **static** or **extern** storage class specifiers.

`illegal struct/union reference`

An object that that was not a pointer to a structure or a union was used where such a pointer was expected.

`illegal to cast a string inside an initialization`

It is illegal to apply the cast operator to a character string used in an initialization.

`illegal to cast to a function`

An object cannot be cast to a function type.

`illegal to cast to an array`

An object cannot be cast to an array type.

`illegal to define defined`

It is illegal to define the identifier defined as a macro. The identifier defined is used as an operator in the expression that follows a **#if** or **#elif** directive.

`illegal to take the size of a function`

The **sizeof** operator cannot be applied to a function type.

`illegal to undefine defined`

It is illegal to undefine the identifier **defined**. The identifier **defined** is used as an operator in the expression that follows a **#if** or **#elif** directive.

`illegal to use & on a bit field`

It is illegal to take the address of a bit field.

`illegal type combination`

An illegal combination of type specifiers appeared together in a declaration.

`illegal type pointer`

In a declaration, an asterisk was followed by a type specifier other than **const** and/or **volatile**.

`illegal #undef of predefined macro` *`identifier`*

The given predefined macro cannot be undefined. It is illegal to undefine the predefined macros: __**DATE**__ , __**FILE**__ , __**FLOAT**__, __**INT**__ , __**LINE**__ , __**PCRELL**__ , __**SIERRA**__ , __**STDC**__ , and __**TIME**__.

`illegal use of &`

The address operator (&) was applied to an object whose address cannot be taken.

`illegal use of & on register type`

It is illegal to take the address of an object declared with the **register** storage class specifier.

`illegal use of member` *`identifier`*

The dot (.) (or arrow (–>)) operator was applied to a structure (or pointer to a structure) that does not contain the given member, and the member is defined in more than one other structure in current scope.

`illegal use of void pointer`

Arithmetic cannot be performed on a pointer to **void**. The size of a **void** object is undefined.

`illegal void declaration`

An object cannot be declared to be of type **void**.

`incompatible operand types, op CAST`

The object that is being cast cannot be converted to an object of the type of the cast.

`incompatible operand,` *`type`*`, op BOOLEAN`

It is illegal to apply the specified Boolean operator to the given *type*.

`incompatible operand, type, op operator`

The given unary *operator* cannot be applied to the given *type*.

`incompatible operands, type and type, indent() arg position`

The argument at the specified position in the function call is not assignment compatible with the type specified in the prototyped function declaration. When passing arguments to a prototyped function, the rules that apply are the same as those that apply when using the assignment operator.

`incompatible operands, type and type, op operator`

It is illegal to apply the given *operator* to the given *types*.

`input token too long`

The length of a preprocessor token (identifier or numeric constant) exceeds 256 characters.

`integer divide by zero`

An integer division by zero occurred while doing integer constant folding.

`invalid argument to #include directive`

The argument to the **#include** directive after macro expansion must be of the form "*filename*" or <*filename*>.

`invalid line number for #line directive`

A **#line** directive must be followed by a decimal constant and an optional double quoted filename.

`identifier is missing from formal parameter list`

The given *identifier* appears in the argument declaration list, but does not appear in the formal parameter list.

`label identifier is missing`

The label *identifier* referred to in a **goto** statement does not appear in the current function.

`leading comma inside function call`

A comma cannot appear in front of the first argument in a function call, nor can it appear in a function call with no arguments.

`live variable analysis stack overflow`

There were more than 50,000 references in the current function to register candidate variables. Simplify the function or reduce the number of register candidates using the **`-Or0`** or **`-Or1`** flags to avoid the problem.

`loops/switches are nested too deep`

Loops cannot be nested deeper than 25 levels and **switch** statements cannot be nested deeper than 20 levels.

`macro definition too long`

A macro definition exceeds 4096 characters.

`macro expansion too long`

The expansion of a macro cannot exceed 120,000 characters.

`memory allocation request too large: code` *`number`*

A memory request by the compiler exceeds the maximum block size available on the host machine. The code *number* is an internal code specifying the location of the memory request in the compiler.

`missing #endif`

A **#if**, **#ifdef**, or **#ifndef** preprocessor directive is not followed by a matching **#endif** directive.

`missing newline after preprocessor directive`

EOF (end of file) was reached before the newline that completes a preprocessor directive.

`missing or illegal argument to #define directive`

The argument to a **#define** preprocessor directive is missing or is not a valid C identifier.

`missing or invalid argument to #undef directive`

The argument to a **#undef** preprocessor directive is missing or is not a valid C identifier.

`missing or invalid macro name for -`*`flag`* `flag`

The argument to a **`-D`** or **`-U`** flag is missing or is an invalid C identifier.

`missing or invalid preprocessor directive`

The token that follows the pound sign (#) used to begin a preprocessor directive is not a valid C identifier.

`missing right parenthesis in argument list for macro` *`ident`*

The right parenthesis needed to terminate a macro argument list is missing from the given macro definition.

`multiple decimal points in constant`

A floating-point constant contains more than one decimal point.

`newline inside character constant`

An unescaped newline appears inside a character constant.

`newline inside character string`

An unescaped newline appears inside a character string.

`newline inside filename`

A newline character appears in the filename of a **#include** directive.

`newline or EOF in character constant`

An unescaped newline or **EOF** (end of file) appears inside a character constant.

`newline or EOF inside string`

An unescaped newline or **EOF** (end of file) appears inside a string.

`operand must be a nonconst lvalue, op` *`operator`*

The operand of the increment (++) or decrement (– –) operator must be modifiable.

`operand must have scalar type, op CAST`

It is illegal to cast aggregate types, such as arrays and structures.

`parameters illegal in a function declaration`

A function declaration must not contain parameters.

`parser stack overflow`

A statement is too complex for the compiler to process. To correct the problem, break the statement into simpler statements.

`pointer illegal in switch`

The expression following the **switch** keyword must evaluate to an integer type.

`pointer to a function can only be assigned, op` *`operator`*

A pointer to a function cannot be operated on other than by an assignment operator.

`preprocessor error` *`number`* `(internal)`

An internal error has occurred in the preprocessor section of the compiler. Please submit a problem report on TI's web site.

`prototype declarations are nested too deep`

Function prototype declarations can be nested up to ten levels deep.

`redeclaration of` *`identifier`*

The given *identifier* is declared more than once.

`redeclaration of formal parameter` *`identifier`*

A formal parameter is declared more than once in the argument declaration list.

`redeclaration of label` *`identifier`*

The given label appears more than once in a function.

`redeclaration of` *`identifier`* `(prototype mismatch)`

The function *identifier* was declared or defined with a prototype argument list that does not match the prototype argument list of an in-scope declaration of the same function.

`redeclaration of tag` *`identifier`*

The given structure, union, or enumeration tag is multiply defined.

`reference to macro` *`identifier`* `is nested too deep`

Macros cannot be nested deeper than 32 levels.

`return operand and function type are incompatible`

The type of the object being returned and the type of the function are incompatible.

`size of operand unknown, op SIZEOF`

The **sizeof** operator is applied to an object or type of unknown size. This can occur when a dimension of an external array is left unspecified.

`size of struct/union` *`identifier`* `is unknown`

The size of the given structure or union is not known; thus, a structure or union assignment cannot be made.

`size of struct/union is unknown`

The size of a structure or union is not known; thus, a structure or union assignment cannot be made.

`size of struct/union member` *`identifier`* `is unknown`

The size of the given structure or union member is unknown.

`size of struct/union/enum` *`identifier`* `is unknown`

The size of the given structure, union, or enumeration type is unknown.

`size of struct/union/enum unknown`

The size of a structure, union, or enumeration type is unknown.

`struct/union/enum definition illegal inside prototype`

A structure, union, or enumeration type cannot be defined in the parameter list of a function prototype declaration.

`struct/union member` *`identifier`* `is undefined`

The given *identifier* is not defined as a member of an in-scope structure or union.

`struct/union pointer (not type) required`

The left operand of the arrow (–>) operator is a structure type instead of a structure pointer.

`struct/union pointer required by nonunique member` *`identifier`*

The given member appears at different offsets in more than one structure in the current scope, and the left operand of the arrow (–>) operator is not a structure pointer.

`struct/union reference illegal in switch`

The expression following the **switch** keyword must evaluate to an integer type.

`struct/union reference must be addressable`

The compiler must be able to take the address of the object to the left of the dot (.) operator.

`struct/union type (not pointer) required`

The operand to the left of the dot (.) operator is a structure pointer instead of a structure type.

`struct/union type required by nonunique member` *`identifier`*

The given member appears at different offsets in more than one structure in the current scope, and the operand to the left of the dot (.) operator is not a structure type.

`structures are nested too deep`

Structures cannot be nested deeper than 50 levels.

`syntax error`

The code is grammatically incorrect.

`syntax error in #if or #elif expression`

The expression following a **#if** or **#elif** directive is grammatically incorrect.

`syntax error in formal parameter list for macro` *`identifier`*

The formal parameter list of the given macro contains a grammatical error.

`too many errors: compiler exiting`

The compiler exits after 100 errors have been reported.

`too many files open, can't include` *`file`*

The given *file* cannot be included because the system limit on the number of files that can be simultaneously opened has been exceeded. If possible, increase the system limit on the number of files that can be opened simultaneously.

`too many formal parameters for macro` *`identifier`*

The parameter list for the given macro contains more than 31 parameters.

`too many include files`

More than 255 files were included in the current source file.

`too many initializers`

A declaration contains more initializers than there are objects to initialize.

`too many initializers at { level` *`number`* `}`

An aggregate declaration contains more initializers than there are objects to initialize at the specified brace level.

`type-name cannot be initialized`

A **typedef** declaration cannot contain initializers.

`type-name is too complex`

The **typedef** declaration is too complex.

`unable to allocate additional memory: code` *`number`*

The memory needed by the compiler exceeds the available memory on the host machine. The code *number* is an internal code specifying the location of the failed memory request. To correct the problem, break up or shorten functions, or reduce memory overhead by running the compiler directly without the command driver, a make utility, user-supplied shells, or add more memory to your system.

`identifier` `undefined`

The given *identifier* is not currently defined.

`unexpected characters after filename in expansion of macro` *`ident`*

A macro in a **#include** directive does not expand to the form "*filename*" or <*filename*>.

`unexpected characters after preprocessor directive`

Only spaces, tabs, and comments are allowed between the last character of a preprocessor directive and the newline that terminates the directive.

`identifier` `unexpected in declaration`

Two identifiers appeared adjacent to each other in a declaration. Most likely, a comma between the two identifiers is missing or the first identifier is assumed to be a **typedef** name, but is not recognized as such by the compiler.

`unrecognized character,` *`number`* `(hex)`

The character represented by the given hexadecimal *number* is not recognized by the compiler.

`unrecognized escape sequence in character constant`

A character constant following a **#if** or **#elif** directive contains an unrecognized escape sequence.

`unrecognized preprocessor directive 'token'`

The given token is not a legal preprocessor directive.

`unsigned type always >= zero`

An unsigned number is compared to zero using the greater than or equal to (>=) or the less than (<) operator. The test is meaningless because unsigned types are always greater than or equal to zero.

`unterminated comment`

EOF (end of file) was reached before the last comment was closed with the comment delimiter (*/).

`void cannot appear with other prototype arguments`

The **void** type specifier cannot be used with any other arguments in a function declaration or definition.

`void function` *`identifier`*`() cannot return value`

It is illegal to return a value from a function declared as returning **void**.

`void type illegal in switch`

The expression following the **switch** keyword must evaluate to an integer type.

`warning: identifier may be used before it is defined`

An automatic variable may be used before its value is set. This warning message appears only when the **-Xu** flag is set. In most cases, the warning can be ignored because the path by which the variable is used before it is defined will never be taken during actual program execution.

`warning: expression is evaluated but not used`

An expression is evaluated, but the result of the evaluation is not used.

`warning: floating point overflow`

Floating-point overflow occurred during compile-time constant folding of floating-point constants.

`warning: floating point underflow`

Floating-point underflow occurred during compile-time constant folding of floating-point constants.

`warning: function ident() called with too few arguments`

The function was called with fewer arguments than were specified by an in-scope prototyped function declaration.

`warning: function ident() called with too many arguments`

The function was called with more arguments than were specified by an in-scope prototyped function declaration.

`warning: function ident() declared without prototype information`

A function was declared without prototype information (old-style declaration). Message can be generated only when **-Xf2** or **-Xf3** command line flag is specified.

`warning: function identifier() used without declaration`

A function was called without a function declaration in scope. This message can be generated only when the **-Xf1** or **-Xf3** command line flag is specified.

`warning: function returning integer cast to pointer`

A function declared as returning an integer is cast to a pointer. When the return value of a function is cast to a pointer type, quite often the function is not declared previously in the file in which case it is assumed to return an **int**. Not declaring a function that returns a pointer, and casting it when used, causes problems because a function that returns a pointer returns its value in register **a0**

and a function that returns an integer type returns its value in register **d0**. In the situation described, the called function will place its return value in **a0** and the calling function will expect the value to be in **d0**.

`warning: illegal combination of type1 and type2, op operator`

It is illegal to combine the given types *type1* and *type2* using the given *operator*.

`warning: illegal combination of enum types, op operator`

It is illegal to combine different enumeration types across an assignment.

`warning: illegal combination of pointers, op operator`

It is illegal to combine different pointers types using the given *operator*.

`warning: illegal position independent initialization`

Because the address of a position independent object or function is not known until load-time or run-time, it cannot be used for compile-time initialization.

`warning: illegal use of member identifier`

The structure to the left of the dot (.) or arrow (–>) operator does not include the given member.

`warning: incompatible parameter types, indent() arg position`

The parameter at the specified argument position in a nonprototyped function definition does not match the corresponding argument in an in-scope prototyped declaration. Often the problem is the result of the promotion of a **char** or **short int** parameter to an **int** in the nonprototyped definition. For more information, see section **2.11.1.3 Mixing Prototype and Old-Style Declarations**.

`warning: loop invariant capacity exceeded`

The number of invariants detected in a loop exceeds the limit imposed by the compiler's loop optimizer. The loop invariants located before the limit was reached will be processed normally by the optimizer. When the limit is reached, the compiler will sometimes generate better code when invariant optimizations are performed on inner-most loops only (**`-Ol2`** flag).

`warning: return operand does not match function type`

The type of the object being returned and the type of the function do not match.

`warning: statement cannot be reached`

There is no path by which the statement can be reached. The compiler does not generate code for unreachable statements.

`warning: string initializer too long`

The string used to initialize a character array contains more characters than the array can hold.

`warning: struct/union pointer required`

The left operand of the arrow (–>) operator is not a structure or union pointer.

`warning: struct/union type required`

The left operand of the arrow (–>) operator is not a structure or union type.

`warning: too many dimensions for symbolic debugging`

The object file symbol table cannot keep track of more than six levels of array dimensions.

`warning: unnecessary assignment of variable` *`identifier`*

An assignment is made to the given variable, and the variable is not used before it is assigned to again or the end of the function is reached.

`wrong number of arguments for macro` *`identifier`*

The number of arguments in a macro invocation does not match the number of arguments in the macro definition.

`zero or negative array dimension`

Array dimensions must be positive integer values.

Section 3: Assembler

Figures

Tables

3. Assembler

This section describes in detail the Sierra Systems assemblers, **asm68** and **asm68k**. It includes a guide to their usage, a discussion of relevant programming concepts, and a complete reference for assembler options and directives. They were developed by Sierra Systems to support certain Motorola processors and coprocessors and IEEE format floating-point numbers. Under license from Sierra Systems, Texas Instruments has modified this software to support TI BCD floating-point numbers, and support for coprocessors has been removed. Although the software has not been modified to exclude support for processors other than the 68000, the 68000 is the only processor supported by Texas Instruments. The license from Texas Instruments to use these products is restricted to development of software that is targeted to execute only on TI calculators.

3.1. Introduction

The assembler **asm68** was designed to assemble source files generated by the Sierra C™ compiler, **com68**. The second assembler **asm68k** was designed to assemble code written for the Motorola M68000 Resident Structured Assembler. Its directive set has been extended to offer many of the features provided by **asm68**, while remaining compatible with code written for the Motorola assembler.

Either assembler can be used to develop assembly language programs and assemble source files generated by the Sierra C compiler. They both support the entire Motorola 68000 instructions and modes. The main difference between the two assemblers, other than syntax, is that **asm68k** provides macro support and structured control facilities whereas **asm68** does not.

The first five assembler sections and the instruction summary section at the end apply to both of these assemblers. Those subsections and descriptions that pertain to a specific assembler are clearly marked with the name of the assembler to which the text is applicable. Sections **3.6 Asm68 Assembler Directives**, **3.7 Asm68k Assembler Directives**, and **3.8 Asm68k Macros** each apply only to the named assembler.

3.1.1. Overview

The assembler section is targeted at the experienced 68000 assembly language programmer and is not intended to serve as a 68000 reference source (see section **3.1.2 Prerequisite Reading**). This section includes the following information:

- Section 3.2 — a guide to using the assemblers.

- Section 3.3 — a discussion of assembly language programming concepts and source file formats.
- Section 3.4 — an instruction set overview, including a discussion of optimizations.
- Section 3.5 — a comprehensive discussion of effective addressing modes.
- Section 3.6 — a discussion of the **asm68** assembler directives, including an alphabetized reference for each assembler.
- Section 3.7 — a discussion of the **asm68k** assembler directives, including an alphabetized reference for each assembler.
- Section 3.8 — a description of **asm68k** user-defined macros and structured control macros.
- Section 3.9 — an instruction-set summary including a list of all supported instructions and legal size extensions.

3.1.2. Prerequisite Reading

The assembler section assumes a working knowledge of the Motorola 68000 microprocessor and the programming issues that govern it and familiarity with the other documentation supplied with the TI-89 / TI-92 Plus SDK. For information regarding these topics, the following sources should be consulted:

- *M68000 Family Resident Structured Assembler Reference Manual*, M68KMASM/D11
- *M68000 Family Programmer's Reference Manual*, M68000PM
- *M68000 8-16-32-Bit Microprocessors User's Manual*, MC68000UM/D
- *MC68881/MC68882 User's Manual*, MC68881UM/AD
- *TI-89 / TI-92 Plus Developer Guide*
- TI **FLASH** Studio™ (IDE) documentation

3.1.3. Notational Conventions

The notational conventions used for syntactic descriptions in the assembler section are shown in Table 3.1. They apply to all syntactic descriptions, unless otherwise stated.

Syntax	Semantics
{ } (Curly Braces)	Enclosed items are required
[ ] (Square Brackets)	Enclosed items are optional
| (Vertical Bar)	Separated items are alternatives
... (Horizontal Ellipsis)	Preceding item can be repeated zero or more times on the same line
⋮ (Vertical Ellipsis)	Preceding item can be repeated zero or more times on successive lines
`typewriter font`	Programming examples
italics	Item is to be replaced with an actual item

Table 3.1: Notational Conventions

Throughout this section, the assembler directives for **asm68** are shown in lower-case characters, while those unique to **asm68k** are shown in uppercase. This is done merely to aid in recognition and to avoid confusion. Both assemblers, in fact, allow directives to be written in either lowercase or uppercase characters. The **asm68k** directives shown in lowercase and beginning with a period (.) are extensions to the Motorola M68000 Resident Structured Assembler. The additional directives are taken from **asm68** and are needed to allow **asm68k** to support the assembly code generated by the Sierra Systems compiler. When a directive is referenced from within a common part of the document, the names of both assemblers' directives are shown, separated by a slash (e.g., .**include / INCLUDE** where **.include** is the **asm68** directive and **INCLUDE** is the **asm68k** directive).

3.2. Invocation

This section describes the command line syntax for invoking the Sierra Systems assemblers. It also includes descriptions of the command line options, file naming conventions, and file inclusion environment variables.

Typically, the TI **FLASH** Studio will handle all invocations of the assembler, using the correct command line flags required to produce TI-89 / TI-92 Plus apps or ASMs. The following discussion of the command line syntax and flags is included for developers who may wish to use either assembler directly from the command line or create their own makefile.

3.2.1. Command Line Syntax

The syntax used to invoke the Sierra Systems assemblers, **asm68** and **asm68k**, is shown below. Option flags are case-sensitive.

```
asm68 [option] . . .    [file] [option] . . .
asm68k [option] . . .   [file] [option] . . .
```

The assembler option *option* specifies zero or more command line option flags. A full description of the assembler flags is provided in section **3.2.2 Command Line Flags.**

The file *file* specifies an assembly source file. If omitted, the input assembly source is read from the standard input stream, **stdin**.

As indicated, the command line arguments consist of option flags and the name of the file to be assembled. All arguments are optional and can appear in any order (except for the **-g** flag and its associated file arguments as noted in section **3.2.2 Command Line Flags**).

3.2.2. Command Line Flags

Command line option flags are used to control the assembler's behavior. This section provides a summary of the available option flags. However, it is highly recommended that you only use the flags as shown in the sample assembler invocations included in the TI-89 / TI-92 Plus SDK files. Other flag combinations may produce output incompatible with the TI-89 / TI-92 Plus data objects. Numerical arguments can be specified in octal (leading 0), decimal, or hexadecimal (leading 0x or 0X); they cannot include any symbols or expressions. Whitespace is permitted between a flag and its argument unless the argument is optional. Multiple flags can follow a single hyphen (-) provided only the final flag takes an argument. Unless otherwise noted, option flags can be placed on the command line in any order. An action specified by an option flag can be undone (returned to the default state) by prefixing the option flag with the **-Z** flag. When two option flags specify conflicting actions, such as an option flag with and without the **-Z** prefix, the latter option flag is used.

The following are the syntax and description of each command line flag:

-a *size* — Specify the address bus size of the target processor (must be 68000 for TI-89 / TI-92 Plus data objects). The bus size *size* is specified in bits. If the **-a** flag is omitted, the address bus size is set according to the selected target processor (see **.opt / OPT** directive). Knowing the size of the address bus is necessary only when relocation hole compression is to be performed (see **-h** flag).

-A Alert the user that the following error messages are from the assembler. The message, “Assembler Errors,” is printed before the first error message. The message separates compiler messages from assembler messages when the compiler and assembler are invoked sequentially.

-c Generate object files that are fully compatible with the AT&T UNIX System V COFF specification. When the **-c** flag is not specified, the **.text**, **.data**, and **.bss** sections begin at relative address zero; comm variables include alignment information; and relocation displacements are not required to fit in their associated relocation holes. This flag is incompatible with the **-6** flag.

-C Cause symbols that are declared **.extern / XREF** but not otherwise referenced in the file not to be entered into the symbol table. In the absence of the **-C** flag, **.extern / XREF** symbols always appear in the symbol table possibly causing unnecessary modules to be pulled in from a library file. The **-C** flag basically causes **.extern / XREF** symbols to behave the same as extern symbols in the C language.

-d *symbol* [= *value*]
Define a local symbol *symbol* with value *value*. If *value* is omitted, the symbol is assigned a value of 1.

-D *symbol* [= *value*]
Define a global symbol *symbol* with value *value*. If *value* is omitted, the symbol is assigned a value of 1.

-e Disable the **.echo** directive. (**asm68**)

-E Require that undefined labels be declared external (see **.xref** directive). Failure to do so will result in an error. By default, undefined labels are assumed to be external. (**asm68k**) Assume that undefined labels are external. By default, undefined labels must be declared external (see **XREF** directive). If the **.opt** directive (not **OPT**) is specified (as it would be in compiler generated assembly code), the **asm68k** assembler emulates the **asm68** assembler as it pertains to undefined labels — undefined symbols are assumed to be external and the **-E** flag necessitates that undefined labels be declared external.

-g *file . . .* Group multiple source files. When specified, the **-g** flag must be the last option on the command line; all files that follow are concatenated to produce a single source file. Output file names are based on the name of the first file that is specified.

-h *file* Select the source file for relocation hole compression. The **-h** flag is used during the first assembly pass. The name of the object file and the command line flags with which it is created are recorded in the hole compression input file *file*. This file, which by convention has a **.hci** extension, is used by **link68**.

-H *file* Perform relocation hole compression on the source file. The **-H** flag is used during the second assembly pass. The information in the hole compression output file *file*, which is created by **link68**, is used to determine which relocation holes can be compressed. No other command line arguments are used during this pass.

-i *file* Include the contents of the file *file* at the current location in the command line. The **-i** flag cannot be used inside an included file; however, multiple occurrences of this flag can appear on the command line.

-I *path* Add the path *path* to the search list used to locate included files (see **.include / INCLUDE** directive). Directories specified with the **-I** flag are searched after the current directory has been searched and before the standard include directories (see section **3.2.4 Environment Variables**).

-k Remove (kill) the object file if any errors are detected.

-K Do not generate an object file.

-l Do not generate a listing file.

-L Generate line number information to allow source-level debugging of assembler source. When the **-L** flag is specified, all debugging directives, except for the **.type / TYPE** directive, are ignored (see sections **3.6.6 Asm68 Debugging Directives** and **3.7.6 Asm68k Debugging Directives**). The **.type / TYPE** directive can be used to specify type information for any label.

-n Do not put line numbers in the listing file.

-o *file* Set the name of the output object file to *file*. If the **-o** flag is not specified, the object file has a **.o** extension and the base name is derived from the name of the source file (see section **3.2.3 File Name Conventions**).

-O[*address*] Set the default section to be an absolute, text-type section with base address *address*. If *address* is omitted, the value zero is used. If the **-O** flag is not specified, the default section is **.text**.

-OPT *option*[,*option*] . . .
Set assembler options. The **-OPT** flag allows assembler options to be specified on the command line (see **.opt / OPT** directive).

-p[*length*] Set the page length used in the listing file to *length*. If *length* is omitted or is zero, pagination is disabled; otherwise, the length must be at least 10. The default page length is 65 lines.

-q Do not require sections to begin on quad boundaries, and do not pad sections to quad boundaries. The **-q** flag is provided to allow the generation of code that can be patched into linked executable files; it should not be used otherwise.

-Q Do not print the Sierra Systems copyright banner.

-r Write error messages to the standard error stream, **stderr**. This is useful only when the input assembly source is read from the standard input stream, **stdin**. The **-r** flag is ignored if an input source file is specified.

-s { x | a[c] | v[c]| c }

Specify which symbols are to be included in the listing file symbol table. The default setting is **a**.

- **x** Do not generate a symbol table listing.
- **a** Generate an alphabetically ordered listing.
- **v** Generate a numerically ordered listing.
- **c** Include compiler local symbols in the listing.

-S { x | l | c }

Specify which symbols are to be included in the object file symbol table. The default setting is **l**.

- **x** Exclude local symbols.
- **l** Include local symbols.
- **c** Include local symbols and compiler local symbols.

-t[*width*] Set the tab width used in the listing file to *width*. If *width* is omitted, or is zero or one, spaces are used instead of tabs; this is the default behavior.

-T Generate a transcript of the input assembly source when it is read from the standard input stream, **stdin**. The transcript file has a **.trn** extension and the base name is derived from the name of the object file (see section **3.2.3 File Name Conventions**).

-u Print usage information, then exit.

-w[*width*] Set the page width used in the listing file to *width*. If *width* is omitted, the page width is set to 132. The minimum width that can be specified is 40. The default page width is 80.

-W Suppress all warning messages.

-x Truncate lines in the listing file at the specified page width (see **-w** flag). The default is to wrap lines that extend beyond the selected page width.

-y *file* Set the name of the listing file to *file*. If the **-y** flag is not specified, the listing file has a **.lst** extension and the base name is derived from the name of the object file (refer to section **3.2.3 File Name Conventions**).

-Y *file* Write error and warning messages to the error file *file*. By default, all such messages are written to the standard error stream, **stderr**.

-Z {6|a|A|c|C|e|E|h|k|K|l|L|n|o|p|q|Q|r|t|T|w|W|x|y|Y} . . .

Returns the specified options to their default conditions.

-6 Not supported by Texas Instruments, however, **-6** is recognized as a reserved flag by the assembler.

3.2.3. File Name Conventions

Each type of file has an associated default file extension as shown in Table 3.2. These extensions are used to determine the names of output files that have not been explicitly specified. When left unspecified, the names of output files are derived from the names of other files as described below.

File Type	File Extension
source	.s
object	.o
listing	.lst
transcript	.trn

Table 3.2: Default File Extensions

If the name of the object file is not specified (see **-o** flag), it is derived from the name of the source file. This is done by adding the **.o** extension to its base filename. Since the full path name is not used, the object file is written in the current directory, regardless of the location of the source file.

If the name of the listing file is not specified (see **-y** flag), it is derived from the name of the object file. This is done by adding the **.lst** extension to its base filename. The path component of the name is preserved so that the listing file resides in the same directory as the object file.

If the assembly source is read from the standard input stream, **stdin** (i.e., no source file was specified), then a transcript file can be generated (see **-T** flag). The name of this file is derived from the name of the object file in the same manner as the name of the listing file described above, except that the replacement extension is **.trn**. In the absence of a source file, the derivation of the object file's name is based on the name of the assembler (e.g., **asm68.o**), and the listing file is written to the standard output stream, **stdout**.

3.2.4. Environment Variables

The environment variables **INCLUDE68** and **SIERRA** can be used to control inclusion of assembly source files (see **.include / INCLUDE** directive). The

INCLUDE68 variable can be used to specify multiple directories by assigning it a semicolon-separated list of paths. The **SIERRA** variable is typically set to the directory under which the Sierra Systems software has been installed. When a

source file is included, the assembler will attempt to locate it in (or, relative to) the following directories in the specified order:

1. The current directory.
2. Directories specified with the **`-I`** command line flag.
3. Directories specified by the environment.

The directories specified by the environment are determined in the following manner. If the **INCLUDE68** variable is defined, the paths it specifies are used; otherwise, the environment search path is the **SIERRA\include** subdirectory.

3.2.5. Invocation Examples

The following examples demonstrate how to invoke the assemblers. The assembler **asm68** is used in the examples. Path names are specified using MS-DOS syntax.

```
asm68 -o ..\obj\driver.o -I..\incl driver.s -W
```

This example assembles the file driver.s. The sibling directory **incl** has been specified as a location to search for included files. The object file, as well as the listing file driver.lst, is created in the sibling directory **obj**. Warning messages have been suppressed with the **`-W`** flag.

In the following example, the assembly source is read from the standard input stream, **stdin**:

```
asm68 -l -T         (or, asm68 -lT)
```

The object file is named **asm68.o**, while the listing file has been suppressed with the **`-l`** flag. The **`-T`** flag directs the assembler to create a transcript file with the name **asm68.trn**.

3.3. Assembly Language

Assembly languages provide an efficient means of developing and maintaining machine-level programs. This section describes the format of assembly language source files, as well as the major concepts necessary to write them.

3.3.1. Overview

The two assemblers, **asm68** and **asm68k**, translate assembler source to object code. This translation is performed using a single pass of the assembler source file. After the source has been completely scanned, forward references are resolved. Any references that cannot be resolved (e.g., references to symbols defined in other files) are left for the linker to resolve.

This section describes the format of assembler source files, management of object code sections, symbol definition and usage, and symbolic expression construction. The format of the object files that are produced is UNIX System V COFF (see section **1.4 Object File Format**).

3.3.2. Assembler Statements

An assembly language program consists of statements that are used to generate machine instructions, control the behavior of the assembler, and provide documentation. The assemblers **asm68** and **asm68k** have different statement syntaxes, which are described below.

3.3.2.1. Statement Syntax (asm68)

The syntaxes for the various types of assembler statements accepted by **asm68** are summarized in Table 3.3. Any of the fields shown can begin in any column, provided they observe the specified syntax. Whitespace (i.e., spaces and tabs) is used to separate the various fields, although no whitespace is required after the label field or before the comment field. Whitespace is permitted within any of the fields.

Statement	Syntax
Label	*label*: [; *comment*]
Instruction	[*label*:] *instruction* [*operand*] [; *comment*]
Directive	[*label*:] *directive* [*operand*] [; *comment*]
Assignment	[*label*:] *symbol* { = | == } *expression* [; *comment*]
Comment	; *comment*

Table 3.3: Statement Syntax (asm68)

Instruction statements direct the assembler to generate object code for the 68000 instructions (see section **3.4 Instruction Set**). Directive statements either modify the behavior of the assembler or direct the assembler to generate object code (see section **3.6 Asm68 Assembler Directives**). Assignment statements associate constant values with symbols (see section **3.3.5.3 Symbol Assignment**). Comment statements are ignored by the assembler; they allow source files to contain explanatory text. The maximum length of any statement is 256 characters.

3.3.2.2. Statement Syntax (asm68k)

The syntaxes for the various types of assembler statements accepted by **asm68k** are summarized in Table 3.4. Labels can begin in any column, provided they are followed by a colon; otherwise, they must begin in column one. Instruction and directive mnemonics, as well as macro invocations, cannot begin in column one.

Statement	Syntax
Label	*label*[:] [! | ; *comment*]
Instruction	[*label*[:]] *instruction* [*operand*] [[! | ;] *comment*]
Directive	[*label*[:]] *directive* [*operand*] [*comment*]
Equate	*label*[:] *equate operand* [*comment*]
Macro	[*label*[:]] *macro* [*operand*] [[! | ;] *comment*]
Comment	{ * | ; } *comment*

Table 3.4: Statement Syntax (asm68k)

Whitespace (i.e., spaces and tabs) is used to separate the various fields of a statement; it can be used within the operand field as part of a character constant or in the comment field. An exclamation point (!) or a semicolon (;) is required at the beginning of the comment field when the preceding operation has optional operands that have been omitted. The asterisk (*) or semicolon (;) that is used to begin a comment statement can appear in any column.

Instruction statements direct the assembler to generate object code for the 68000 instructions (see section **3.4 Instruction Set**). Directive statements either modify the behavior of the assembler or direct the assembler to generate object code (see section **3.7 Asm68k Assembler Directives**). Macro statements allow common sequences of code to be easily duplicated (see section **3.8 Asm68k Macros**). Comment statements are ignored by the assembler; they allow source files to contain explanatory text. The maximum length of any statement is 256 characters.

3.3.3. Character Set

The assembler recognizes the following characters:

- The alphabetical characters A – Z and a – z.
- The numerical characters 0 – 9.
- The whitespace characters space, tab, and newline
- The remaining printable ASCII characters:

  ! " # $ % & ' ( ) * + , - . / :

  ; < = > ? @ [\ ] ^ _ ` { | } ~

Any other characters will cause an error unless they appear in a comment field or string. By default, alphabetical characters are case-sensitive. The **OPT NOCASE** directive (supported by **asm68k**) can be used to cause case insensitivity. When case insensitivity is specified, symbol names are converted to lowercase before entry into the symbol table.

3.3.4. Sections

A section is a block of object code whose base is associated with an address. Sections with fixed base addresses are called *absolute*; those with repositionable bases are called *relocatable*. Relocatable sections are fixed at an absolute address during linkage. Each relocatable section must have a name; names are optional for absolute sections.

The remainder of this section describes the various types of sections and the methods used to manage them.

3.3.4.1. Section Types

There are three types of sections: text-type, data-type, and BSS-type. Text-type sections contain read-only data. Data-type sections contain initialized read/write data. BSS-type sections contain uninitialized read/write data.

There are three special relocatable sections, one of each of the three types. They are called **.text**, **.data**, and **.bss**. These sections have special significance in the System V COFF file format (see section **1.4 Object File Format**). The Sierra C compiler generates these three sections. Code is placed in the section **.text**, initialized data is placed in the section **.data**, and uninitialized data is allocated space in the section **.bss** (see **.comm** and **.lcomm** directives).

> **Note:** TI-89 / TI-92 Plus apps and ASMs also have a **.const** section. See chapter **7. Flash Application Layout** in the TI-89 / TI-92 Plus Developer Guide for information on the use and initialization of the TI-89 / TI-92 Plus sections.

3.3.4.2. Creating Sections

Special directives are used to create and activate sections (see sections **3.6 Asm68 Assembler Directives** and **3.7 Asm68k Assembler Directives**). All object code that is generated by instructions and directives is placed in the most recently created or activated section. When a section has been completed, its size is padded (if necessary) to a quad boundary with the current fill value (see **.opt fillval / OPT FILLVAL**). Prior to the issuance of the first section directive, the relocatable, text-type section **.text** is used. If the **-0** flag is specified, however, the default section is an absolute, text-type section beginning at the specified address.

3.3.4.3. Location Counter

Each section has a location counter, which maintains the current position within the section, i.e., the address at which object code will be placed next. This address is absolute in an absolute section and is section-relative in a relocatable section.

The location counter allows a resumed section to continue as if there had been no interruption: additional object code is placed immediately following the last code that was generated in that section. It also allows instructions and directives to reference the address at which they are located. The location counter can be referenced in integer expressions with a period (`.`) (**asm68**) or an asterisk (`*`) (**asm68k**).

3.3.4.4. Structure Templates

Structure template sections are used to define labels suitable for structure field references. They are merely dummy sections — they cannot contain any object code, and they are not included in the output object file. See the **.struct / OFFSET** directives for a complete description of their usage.

3.3.5. Symbols

Symbols are used to label addresses and represent constants. They can be either absolute or relocatable. The value of an absolute symbol is fixed, while that of a relocatable symbol is dependent upon the section in which it is defined. During assembly, the value of a relocatable symbol is an offset from the base of a relocatable section; it becomes absolute during linkage (after relocatable sections have been bound to absolute addresses).

By default, the scope of a symbol is *static* (i.e., local to the file in which it is defined). The scope of a symbol can be extended to *external* with the **.xdef / XDEF** directive; this allows it to be referenced from within any other source file that declares it to be external (see **.xref / XREF** directive). The term *local* is reserved for describing scopes that are limited to a subset of a source file (e.g., a label local to a macro invocation).

The remainder of this section describes the syntax of symbol names and the methods available to define symbols, set their values, and modify their scopes.

3.3.5.1. Symbol Syntax

The syntax used for constructing symbol names differs between the two assemblers. When using **asm68**, symbol names can be composed of the following characters:

- The alphabetical characters A – Z and a – z.
- The numerical characters 0 – 9.
- The underscore (_).

They cannot begin with a numerical character. When using **asm68k**, symbol names can be composed of the following characters:

- The alphabetical characters A – Z and a – z.
- The numerical characters 0 – 9.
- The underscore (_).
- The period (.).
- The dollar sign ($).

They cannot begin with a numerical character or the dollar sign. For certain types of symbol names, there are exceptions to these rules. They are clearly noted where applicable.

Symbol names are case-sensitive by default. The **OPT NOCASE** directive can be used to render symbol names case-insensitive (**asm68k**). When symbol names are case-insensitive, they are converted to lowercase before entry into the symbol table. There is no explicit maximum length for symbol names. Their length is limited only by the maximum length of a source input line, which is 256 characters. All characters of a symbol name are significant.

3.3.5.2. Labels

A label is a symbol that is used to represent an address in a section. Labels are defined as shown in the description of the various statement syntaxes (see section **3.3.2 Assembler Statements**). A label is either absolute or relocatable; this property is inherited from the section in which it is defined.

When a label is defined, it is assigned the value of the current section's location counter (i.e., the current object code address). In the following code fragment, the symbol loop is assigned the address of the move instruction:

```
                jsr             init
loop:           move.l          (a0)+,(a1)+
                cmpa.l          #end,a0
                bne             loop
```

This value is either absolute or relocatable, depending on the active section. As discussed earlier, relocatable sections are bound to absolute addresses during linkage. At this time, relocatable labels become absolute.

By default, the scope of a label is static (i.e., local to the file in which it is defined). This scope can be extended to other files by defining the label to be external (see **.xdef / XDEF** directive). When using **asm68k**, it is also possible to limit the scope of a label to a particular subset of a source file. A local label is defined by prefixing its name with a period (.) (with the **OPT LLBL** directive in effect). Ordinary (i.e., non-local) labels delineate unique local label scopes, i.e., each time an ordinary label is defined, a new local label scope is created. This means that the same label name can be used in different local label scopes without conflict. In addition, labels can be defined to be local to macro invocations (see section **3.8.1.4 Local Labels**).

3.3.5.3. Symbol Assignment

Symbols can be assigned arbitrary values by direct assignment. When using **asm68**, assignment is performed with the = and == operators. The right-hand side of the assignment is an absolute or simple relocatable expression (see section **3.3.7.3 Expression Evaluation**) that contains no forward, external, or undefined references. The value of this expression is assigned to the symbol on the left-hand side of the assignment.

Using the = operator prohibits the defined symbol from being assigned a new value, while the == operator permits future assignments. Attempting to assign a value to a symbol whose value has been set with the = operator results in an error.

When using **asm68k**, the **EQU** and **SET** directives perform the same function as the = and == operators, respectively (see section **3.7 Asm68k Assembler Directives**). As with labels, the default scope of symbols defined via assignment is static (i.e., local to the file in which they are defined). The **.xdef / XDEF** directive can be used to define a symbol to be external.

3.3.5.4. Comm and Lcomm Symbols

The final way that symbols can be defined is with the **.comm / COMM** and **.lcomm / LCOMM** directives. These directives are used to define symbols that are referred to as comm and lcomm symbols, respectively. Each such symbol is associated with a block of uninitialized data in the **.bss** section. The primary difference between the two types of symbols is their scope: comm symbols are external and lcomm symbols are static (i.e., local to the file in which they are defined). For syntax, descriptions, and examples of these directives, refer to sections **3.6 Asm68 Assembler Directives** and **3.7 Asm68k Assembler Directives**.

The names for these two types of symbols come from the expressions *common area* and *common block*, which are used in languages such as FORTRAN. These directives were originally used for compatibility with these languages; therefore, they allow symbols to be redefined, provided the size and alignment information are identical. If the **-C** linker command flag is specified at link time, definitions of the same symbol can appear in other source files as well. These directives are now typically used for allocating space for uninitialized data. For historical reasons, the names have remained.

Since comm symbols can be defined in multiple source files, they cannot be allocated by the assembler as lcomm symbols are. They are instead allocated during linkage after all common definitions have been resolved.

3.3.5.5. Undefined Symbols

If a symbol is referenced in a source file, but is not defined in that file (by label, assignment, etc.) and has no external declaration, it is considered to be undefined. When using **asm68**, any undefined symbols are assumed to be external. When using **asm68k**, an error is generated for each undefined symbol that is found. These behaviors can be reversed using the **-E** command line flag. Furthermore, if the **.opt** (not **OPT**) directive is used in a file assembled by **asm68k**, all undefined symbols are assumed to be external. The behavior switch based on the detection of the **.opt** directive is needed to allow compiler generated assembly code to be assembled by **asm68k**.

For both assemblers, undefined symbols are defined to be external in the output object file and are shown as undefined in the listing file. The only difference in behavior is whether or not an error message is generated.

3.3.5.6. Compiler Locals

The Sierra C compiler creates labels for data addresses and branch destinations; these labels are static (i.e., local to the files in which they appear) and are commonly referred to as *compiler locals*. Compiler locals are distinguished from other labels by the naming convention they observe: the letter ‘ L ’ followed by an arbitrary number of decimal digits. This convention can be modified to include a leading underscore by specifying the **.opt uloc** directive.

Compiler locals are typically of no interest and, by default, are excluded from the symbol tables in the object and listing files. This exclusion can be overridden with the **-S** and **-s** command line option flags, respectively.

3.3.5.7. Floating-Point Symbols

A floating-point symbol is defined by assigning it a floating-point constant. Since the floating-point assignment operator (#=) in **asm68** and the **FEQU** directive in **asm68k** are not supported by Texas Instruments, the **.double** directive should be used to create floating-point symbols. See the **.double** directive in sections **3.6.7 Asm68 Directive Reference** and **3.7.7 Asm68k Directive Reference** for examples.

3.3.6. Constants

A constant is an invariant quantity, which can be specified in any of the following formats: integer, character, and floating point. Each type has its own value ranges; an error is generated if a range is exceeded.

3.3.6.1. Integer Constants

Integer constants can be specified in binary, octal, decimal, and hexadecimal. The base of an integer is determined by its leading character(s) as shown in Table 3.5. The syntax for specifying the base of an integer constant is different for the two assemblers. Any integer constant can be preceded by the unary minus (-) operator.

Radix	Prefix (asm68)	Prefix (asm68k)
Binary	%	%
Octal	@ or 0	@ or 0†
Decimal	1 – 9	0† – 9
Hexadecimal	$ or 0x or 0X	$ or 0x or 0X

† A leading zero identifies a decimal constant unless the **.opt** (not **OPT**) directive is used in a file assembled by **asm68k**, in which case, a leading zero specifies an octal constant. The behavior switch based on the detection of the **.opt** directive is provided to allow **asm68k** to be compatible with code originally written for **asm68**.

Table 3.5: Integer Radix Specification

The size of an integer constant can be one, two, or four bytes. The value ranges for each size are shown in Table 3.6. Values are stored using two's complement representation.

Bytes	Signed Range	Unsigned Range
1	-128 – 127	0 – 255
2	-32,768 – 32,767	0 – 65,535
4	-2,147,483,648 – 2,147,483,647	0 – 4,294,967,295

Table 3.6: Value Ranges for Integer Constants

3.3.6.2. Character Constants

Character constants are delimited by single quotes ('). Each character between the single quotes is represented by a byte of data; therefore, byte data is limited to a single character, word data to two, and long-word data to four. If the maximum number of characters is not specified, the bytes in a word or long-word constant are justified according to the assembler being used. The bytes are right-justified within the byte group when using **asm68** (or **asm68** style data directives with **asm68k**). For example:

```
.word 'a'              ; produces 0061
.long 'ab'             ; produces 00006162
```

When using **asm68k**, the bytes are left-justified:

```
DC.W  'a'              ; produces 6100
DC.L  'ab'             ; produces 61620000
```

A single quote (') is represented by a pair of single quotes (' ') when using **asm68k**, and by an escaped single quote (\') when using **asm68**. Furthermore, if the **.opt** (not **OPT**) directive is used in a file assembled by **asm68k**, escaped characters are allowed in a character constant appearing in the effective address field of an instruction. The behavior switch based on the detection of the **.opt** directive is needed to allow compiler generated assembly code to be assembled by **asm68k**. The escaped characters recognized by **asm68** (and **asm68k** in the presence of the **.opt** directive or in an **asm68** style data directive) are shown in Table 3.7.

Syntax	Description
\a	bell (alert)
\b	backspace
\f	form feed
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\x*h*	hexadecimal constant (*h* is at most 3 hexadecimal digits)
\X*h*	hexadecimal constant (*h* is at most 3 hexadecimal digits)
o	octal constant (*o* is at most 3 octal digits)
\^*c*	<ctrl>-*c* (*c* is an alphabetical character)
c	*c* (*c* is any character excluding escaped characters shown here)

Table 3.7: Escaped Characters

When a character constant is used with the **DC** directive (**asm68k**), it is legal to exceed the character limit. It has the following effect:

```
DC.B        'abc'       ; equivalent to DC.B 'a','b','c'
DC.W        'abc'       ; equivalent to DC.W 'ab','c'
```

When a character constant is used with the **.byte** or **.ascii** directive, it is legal to exceed the character limit by specifying the constant as a string. Strings are delimited by double quotes ("). It has the following effect:

```
.byte "xyz"             ; equivalent to .byte 'x','y','z'
```

Strings are not null terminated (whereas they are in the C language).

3.3.6.3. Floating-Point Constants

Floating-point constants can be specified in real format or in the internal format of TI BCD floating-point data objects.

Real format constants include a decimal point or an exponent or both. The decimal point is designated with a period (.), and the exponent field is introduced with the letter **e** (or **E**). Internally, the constant is converted to the TI BCD floating-point format.

Floating-point constants can be specified directly in the TI BCD floating-point format by specifying their bit patterns in hexadecimal. The prefixes used to introduce a bit pattern are the hexadecimal integer constant prefixes ($, 0x, or 0X). In addition, when using **asm68**, a colon (:) can also be used to introduce a floating-point hexadecimal bit pattern. All TI BCD floating-point data objects are stored in 10 bytes. See the **.double** directive in sections **3.6.7 Asm68 Directive Reference** and **3.7.7 Asm68k Directive Reference** for examples of floating-point constant entry.

3.3.7. Expressions

Expressions can be used almost anywhere an integer argument is required (e.g., effective address arguments, directive arguments, and right-hand sides of assignments). Floating-point expressions are not supported.

The syntax for expressions is identical to that of the C language, except that only a relevant subset of the operators is supported. There is no limit to the number of operands or operators that can appear in an expression, nor is there a limit to the level of parenthesization. Except where noted, expressions can contain forward, external, and undefined references.

Following is a description of the legal operands and operators, as well as a discussion of expression evaluation.

3.3.7.1. Operands

Integer symbols, integer constants, character constants, and the location counter symbol can be used as operands in an expression. Floating-point symbols and constants cannot be used as operands.

3.3.7.2. Operators

All the arithmetic and bitwise operators found in the C language are supported. They are shown in Table 3.8.

Operator	Description
Unary	
~	One's complement
–	Unary minus
Binary	
*	Multiplication
/	Division
%	Modulus
+	Addition
–	Subtraction
<<	Left shift
>>	Right shift
&	Bitwise AND
^	Bitwise exclusive OR
| or !	Bitwise inclusive OR

Table 3.8: Integer Expression Operators

Each assembler has its own operator precedence rules as shown in Table 3.9. Operators listed on the same line have the same precedence; rows are arranged in decreasing order of precedence. Also shown is the associativity of each operator. The rules for **asm68** are identical to those of the C language, while the rules for **asm68k** are identical to those of the Motorola M68000 Resident Structured Assembler.

asm68	
Operator	**Associativity**
( )	left to right
~ –	right to left
* / %	left to right
+ –	left to right
<< >>	left to right
&	left to right
| !	left to right
^	left to right

asm68k	
Operator	**Associativity**
( )	left to right
~ –	right to left
<< >>	left to right
& | ! ^	left to right
* / %	left to right
+ –	left to right

Table 3.9: Operator Precedence/Associativity

3.3.7.3. Expression Evaluation

Expressions are evaluated using the precedence and associativity rules described above. They are always calculated with 32-bit intermediate values, regardless of the size of the result. An expression can be evaluated when it is encountered if it contains no forward or undefined symbol references. Otherwise, the evaluation of the expression is deferred until the source file has been completely scanned so that any forward references can be resolved.

Once an expression has been evaluated by the assembler, it can be classified as one of the following three types: absolute, simple relocatable, or complex relocatable. An expression that evaluates to an absolute value is classified absolute. If the expression has reduced to an absolute offset from an external symbol or from the base address of a relocatable section, then the expression is simple relocatable; otherwise, the expression is complex relocatable. An expression that references multiple external symbols or symbols from different relocatable sections would be classified as complex relocatable. See Figure 3.1 for an example of the different types of expressions.

```
       .xref        sym_x
       sym_a = 18
sym_b:
       .long        0xff << sym_a  ; absolute
       .long        sym_f – sym_b  ; absolute (deferred)
       .long        . + 8          ; simple relocatable
       .long        sym_f – 4      ; simple relocatable (deferred)
       .long        sym_b – sym_x  ; complex relocatable
       .long        sym_b + sym_f  ; complex relocatable (deferred)
sym_f:
```

Figure 3.1: Expression Evaluation

If the expression is relocatable, a relocation entry is generated and placed in the object file so that the evaluation can be performed during linkage. All relocatable expressions are resolved to absolute values after section allocation has been performed (i.e., each relocatable section has been bound to an absolute address). See section **1.4.5.2 Complex Relocation**, for a complete description of how complex relocation is performed.

Note: Complex relocation is not supported when either the **-6** option or **-c** option has been selected. An error is generated when a complex relocatable expression is used with either of these flags.

Note: For users of the Motorola M68000 Resident Structured assembler (**asm68k**): The class of expressions that are classified as complex relocatable has been extended to include any expression that is neither absolute nor simple relocatable. Furthermore, the limitations imposed on their usage in effective addresses have been lifted; they can be used anywhere a simple relocatable expression can be used.

If an expression affects the size of the object code, its value must resolve to an absolute value when it is encountered. Also, expressions used in assignments must resolve to an absolute or simple relocatable expression when encountered. Expressions used elsewhere will typically carry no restrictions. Any restrictions that are applicable to a particular expression are noted with the description of its usage.

If an expression is used as immediate data or an argument to a data directive, and the size of the data is byte (8) or word (16), a “hole too small” warning will be generated by the linker if the expression evaluates to a value outside the range of a signed byte or word, respectively. An unsigned byte or word with the high order bit set is rejected because it would be interpreted as a negative value when used in an effective address as an absolute address or displacement. The warning is sometimes incorrectly generated because the linker has no information as to whether the expression it is evaluating is used as immediate data (warning potentially incorrect) or in an effective address (warning correct).

To get around the warning problem, the linker can be forced to accept expressions that resolve to unsigned values in the range 0 to 0xffff (0xff in the case of a byte) by explicitly masking the entire expression with 0xffff (0xff). For example, where the expression A + B resolves to the value 0x9040, the first statement will generate a linker warning and the second will be accepted:

```
.word       A+B             ; linker warning generated
.word       (A+B)&0xffff    ; ok
```

3.4. Instruction Set

Only the 68000 instruction set is supported by Texas Instruments. Use of unsupported instructions gives unpredictable results. This is especially true of the floating-point instructions. See Table 3.32 in section **3.9 Instruction Set Summary** for a complete list of supported instructions. For detailed information on any instruction, consult the appropriate Motorola reference manual (see section **3.1.2 Prerequisite Reading**). This section describes the accepted instruction syntax, the sizing of instructions, and the optimizations that can be performed.

3.4.1. Syntax

The assemblers **asm68** and **asm68k** recognize the instruction mnemonics that are used by Motorola. Mnemonics can be specified in either all uppercase or all lowercase characters, and can include an optional size qualifier. The recognized mnemonics and corresponding legal size qualifiers are shown in Table 3.32 in section **3.9 Instruction Set Summary**. For restrictions on the placement of instructions imposed by the statement syntax, see section **3.3.2 Assembler Statements**.

3.4.2. Instruction Sizing

The size of an instruction is determined according to the first applicable rule in the following list:

1. The size extension is used, if present.
2. The current default instruction size is used, if legal (see **.opt isize / OPT ISIZE** directive).
3. The instruction's default size is used (see Table 3.32, section **3.9 Instruction Set Summary**).

Figure 3.2 demonstrates how these rules are used to determine the sizes of various instructions. Word size is not legal for the **lea** instruction, and the **jsr** instruction is unsized.

```
movea.l     #array,a0       ; instruction size is long (rule 1)
move        func,d0         ; instruction size is long (rule 2)
.opt        isize = w       ; set default instruction size to word
move        func,d1         ; instruction size is word (rule 2)
lea         (a0,d0),a1      ; instruction size is long (rule 3)
jsr         (a1)            ; not applicable
```

Figure 3.2: Instruction Sizing (asm68)

3.4.3. Instruction Optimization

By default, both assemblers perform instruction optimizations and corrections. This feature can be disabled with the **.opt noiopt / OPT NOIOPT** directive. An instruction is optimized by replacing it with a smaller, faster instruction that has the same functionality (e.g., `add.l #4,d0` can be replaced with `addq #4,d0`). Corrections are performed on instructions when one of the effective addressing modes is incompatible with the instruction. In this case, the instruction is replaced with a closely related instruction in order to produce legal code (e.g., `add.l d0,a0` must be replaced with `adda.l d0,a0`). Instructions that are not corrected will result in an error.

The instruction optimizations and corrections are summarized in Table 3.10. As is shown, optimizations produce faster code with exactly the same results, and corrections produce legal code with the original code's intent.

Instruction		Modified Form	
ADD	<ea>,An	ADDA	<ea>,An
ADD	#<data>,<ea>	ADDI	#<data>,<ea>
ADDI	#<data>,An	ADDA	#<data>,An
ADD	#<qdata>,<ea>	ADDQ	#<qdata>,<ea>
ADDI	#<qdata>,<ea>	ADDQ	#<qdata>,<ea>
ADDA	#<qdata>,An	ADDQ	#<qdata>,An
SUB	<ea>,An	SUBA	<ea>,An
SUB	#<data>,<ea>	SUBI	#<data>,<ea>
SUBI	#<data>,An	SUBA	#<data>,An
SUB	#<qdata>,<ea>	SUBQ	#<qdata>,<ea>
SUBI	#<qdata>,<ea>	SUBQ	#<qdata>,<ea>
SUBA	#<qdata>,An	SUBQ	#<qdata>,An
CMP	#<data>,<ea>	CMPI	#<data>,<ea>
CMP	<ea>,An	CMPA	<ea>,An
CMPI	#<data>,An	CMPA	#<data>,An
CMP	(Ay)+,(Ax)+	CMPM	(Ay)+,(Ax)+
AND	#<data>,<ea>	ANDI	#<data>,<ea>
OR	#<data>,<ea>	ORI	#<data>,<ea>
EOR	#<data>,<ea>	EORI	#<data>,<ea>
MOVE.L	#<bdata>,Dn	MOVEQ	#<bdata>,Dn
MOVE	<ea>,An	MOVEA	<ea>,An

Notation: <ea> = any legal effective address
#<data> = immediate data
#<bdata> = byte immediate data (byte)
#<qdata> = quick immediate data (range 1–8)

Table 3.10: Instruction Optimizations and Corrections

3.5. Effective Addressing Modes

Both assemblers support all effective addressing modes for the 68000 microprocessor. They recognize a flexible addressing mode syntax, while providing the user with full control over addressing mode selection. Additionally, they perform various optimizations and coercions to allow the most efficient addressing possible. This section describes the 68000 effective addressing modes, relevant terminology, the accepted syntax for the addressing modes, and which modes are generated given a particular syntax and set of user options.

3.5.1. Overview

The effective addressing modes supported by the various 68000 family processors are summarized in Table 3.11; also shown are the processors to which each addressing mode is applicable. However, only the 68000 addressing modes are supported by Texas Instruments. For convenience, the effective address notation used in this table and throughout this section is identical to that which is used in the Motorola reference manuals.

The various effective addressing modes can be categorized according to the ways in which they can be used. The following is a list of classifications that are useful for describing restrictions on the use of different addressing modes.

- Data — the addressing mode can be used to refer to data operands.
- Memory — the addressing mode can be used to refer to memory operands.
- Control — the addressing mode can be used to refer to memory operands that do not have an associated size.
- Alterable — the addressing mode can be used to refer to alterable (i.e., writeable) operands.

These terms can be combined to form more restrictive classes of effective addressing modes. For example, a *data alterable* mode is one that is both a data reference and alterable. Table 3.12 summarize the effective addressing modes and the categories to which they belong.

Addressing Mode	Syntax	68000/08/10	CPU32*	68020/30/40*
Register Direct				
Data	Dn	✓	✓	✓
Address	An	✓	✓	✓
Register Indirect				
Address	(An)	✓	✓	✓
Address with Postincrement	(An)+	✓	✓	✓
Address with Predecrement	–(An)	✓	✓	✓
Address with Displacement	(d_{16},An)	✓	✓	✓
Address Register Indirect with Index				
8-Bit Displacement	(d_8,An,Xn)	✓	✓	✓
Base Displacement	(bd,An,Xn)		✓	✓
Memory Indirect				
Postindexed	([bd,An],Xn,od)			✓
Preindexed	([bd,An,Xn],od)			✓
Program Counter Indirect				
with Displacement	(d_{16},PC)	✓	✓	✓
Program Counter Indirect with Index				
8-Bit Displacement	(d_8,PC,Xn)	✓	✓	✓
Base Displacement	(bd,PC,Xn)		✓	✓
Program Counter Memory Indirect				
Postindexed	([bd,PC],Xn,od)			✓
Preindexed	([bd,PC,Xn],od)			✓
Absolute Data Addressing				
Short	(xxx).W	✓	✓	✓
Long	(xxx).L	✓	✓	✓
Immediate	#<data>	✓	✓	✓

*Not supported by Texas Instruments.

Notation:
Dn = Data register
An = Address register
PC = Program counter
Xn = Index register
xxx = Expression
d_8 = Byte displacement
d_{16} = Word displacement
bd = Base displacement
od = Outer displacement
<data> = Immediate value

Table 3.11: Effective Addressing Modes

Addressing Mode	Data	Memory	Control	Alterable
Register Direct				
Data	✓			✓
Address				✓
Register Indirect				
Address	✓	✓	✓	✓
Address with Postincrement	✓	✓		✓
Address with Predecrement	✓	✓		✓
Address with Displacement	✓	✓	✓	✓
Address Register Indirect with Index				
8-Bit Displacement	✓	✓	✓	✓
Base Displacement	✓	✓	✓	✓
Memory Indirect				
Postindexed	✓	✓	✓	✓
Preindexed	✓	✓	✓	✓
Program Counter Indirect				
with Displacement	✓	✓	✓	
Program Counter Indirect with Index				
8-Bit Displacement	✓	✓	✓	
Base Displacement	✓	✓	✓	
Program Counter Memory Indirect				
Postindexed	✓	✓	✓	
Preindexed	✓	✓	✓	
Absolute Data Addressing				
Short	✓	✓	✓	✓
Long	✓	✓	✓	✓
Immediate	✓	✓		

Table 3.12: Effective Addressing Mode Categories

3.5.2. Terminology

The term *base register* refers to the register that is used as the base address in an indexed addressing mode. Any address register or the program counter can be used as a base register. The term *index register* refers to the register that is added to the base address in an indexed addressing mode. Any data register or address register can be used as an index register.

If the base or index register is omitted from an addressing mode, it is said to be *suppressed*. When a register is suppressed, its value is zero in the effective address calculation. Since Texas Instruments only supports the 68000, suppression is not allowed.

The term *displacement* is used to describe either an absolute address or a signed offset from a base address. Any integer expression (see section **3.3.7 Expressions**) can be used to designate a displacement. The following types of displacements are used in the 68000 family effective addressing modes, however, only the displacements allowed on the 68000 are supported by Texas Instruments:

- Byte displacement — d_8
- Word displacement — d_{16}
- Base displacement — bd
- Outer displacement — od
- Absolute short address — (xxx).W
- Absolute long address — (xxx).L

Base and outer displacements can be either a word or a long word. If omitted, they are said to be *null* displacements, which have a value of zero in effective address calculations.

3.5.3. Effective Address Syntax

This section describes how effective addresses are specified in assembly language. The syntax accepted by both assemblers is an extension of the Motorola syntax in that it provides more flexibility in the placement of the effective address operands and permits the use of whitespace (**asm68** only) for increased readability. The accepted syntax is as follows:

- Parentheses are used to designate indirection through a register.
- The comma (,) is used to separate address components.
- The plus sign (+) and the minus sign (–) are used to specify the postincrement and predecrement modes, respectively.
- The pound sign (#) is used to designate immediate data.
- The size qualifiers `.w` and `.l` can be appended to any index register.
- All registers (i.e., data, address, floating-point, and control registers) are specified using Motorola's naming conventions.

- Registers can be specified in either all uppercase or all lowercase characters.
- **sp** (or **SP**) is a synonym for **a7**.
- The hyphen (–) and slash (/) are used to specify register lists; the hyphen is used to designate an ascending register range and the slash is used to separate ranges (e.g., **d3–d7/a2–a4**).
- Whitespace can be used between operands and within expressions (**asm68**).

All operands are required for the addressing modes that apply to the 68000; in the other effective addressing modes, all operands are optional. If a register operand is legally omitted from an addressing mode, it is suppressed (see section **3.5.2 Terminology**). Since Texas Instruments only supports the 68000, suppression is not allowed.

The order of operands within parentheses is significant only if it results in ambiguity. The only such case is when two address registers, of which one or both can be suppressed, are used and neither has a size qualifier or scaling factor. Since it cannot be determined which is the base register, the one on the left is selected arbitrarily. The address register on the right is used as the index register.

For backward compatibility, the old-style displacement notation associated with the 68000 is accepted; the effects of specifying effective addresses using old-style 68000 syntax are discussed below. Table 3.13 shows the four applicable 68000 addressing modes with both old and new syntaxes.

Addressing Mode	Old Syntax	New Syntax
Address Register Indirect with Displacement	d_{16}(An)	(d_{16},An)
Address Register Indirect with Index (8-Bit Displacement)	d_8(An,Xn)	(d_8,An,Xn)
Program Counter Indirect with Displacement	d_{16}(PC)	(d_{16},PC)
Program Counter Indirect with Index (8-Bit Displacement)	d_8(PC,Xn)	(d_8,PC,Xn)

Table 3.13: Displacement Syntax Comparisons

Note: The addressing mode (*expr*,PC) or *expr*(PC) means that *expr* is referenced relative to the program counter; it does not represent a relative reference of *expr* bytes from the PC. For example, (8,PC) references the absolute address 8 via the program counter. To represent a PC-relative reference of 8 bytes from the beginning of the current instruction, the correct syntax is (.+8,PC) or (*+8,PC), depending on the assembler.

3.5.4. Addressing Mode Selection

The assembler performs various optimizations and coercions on eligible effective addressing modes in order to produce more efficient code. The following addressing modes are not affected:

- Data register direct.
- Address register direct.
- Address register indirect.
- Address register indirect with postincrement.
- Address register indirect with predecrement.
- Immediate data.

For all other addressing modes, the mode selection is carried out in three phases. First, coercion to a PC-relative addressing mode is performed, if possible. Using a PC-relative mode typically results in a smaller, faster instruction. This step can be bypassed with the **.opt nopc / OPT NOPC** directive. Second, displacements are sized minimally for the current set of user-specified effective address options. Finally, the mode selection is made.

3.5.4.1. PC-relative Coercion

The effective address specifications that are eligible for PC-relative coercion are those that have a displacement and no base register. This includes the absolute addressing modes, the register indirect modes, and the memory indirect modes. The following criteria must be met for coercion to be performed:

1. The appropriate PC addressing mode must be legal for the instruction.
2. The displacement must not have a size qualifier.
3. The settings of the PC-relative coercion options must allow it.

In the following discussion, **pcb32 / PCB32** is included for completeness, but is not supported by Texas Instruments. The options related to PC-relative coercion (see **.opt / OPT** directive) are the following:

pcb16 / PCB16	**pcf / PCF**	**nopc / NOPC**
pcb32 / PCB32	**pca / PCA**	**nopca / NOPCA**

These settings are used as described below to control the situations in which PC-relative coercion is performed. The primary consideration is whether or not a PC-relative displacement can be computed when its associated effective address is encountered. The computation can be performed if the displacement involves no forward or external references, and the effective address and referenced

location are both in the same relocatable section or are both in absolute sections (see section **3.3.7.3 Expression Evaluation**).

If the displacement can be computed relative to the program counter when it is encountered, then coercion is performed if one of the following two conditions is met:

- The **pcb32 / PCB32** option is in effect.
- The **pcb16 / PCB16** option is in effect and the relative displacement fits in a word.

If the displacement cannot be computed relative to the program counter, then coercion is performed if one of the following three conditions is met:

- The **pcb16 / PCB16** option is in effect and the coercion is permitted by relocation hole compression.
- The **pcf / PCF** option and the **pca / PCA** option are both in effect.
- The **pcf / PCF** option is in effect and neither the displacement nor the reference of the displacement is known to be in an absolute section when the instruction is encountered.

To generate position-independent code, the **pcf / PCF** and **nopca / NOPCA** options should be selected. For more information on these option settings see sections **3.6 Asm68 Assembler Directives** and **3.7 Asm68k Assembler Directives**.

3.5.4.2. Displacement Sizing

If a size qualifier is present, the designated size is used. An error is generated if it can be determined that this size is too small for the displacement. In the absence of a size qualifier, the size of the displacement is based on its value; however, if the value cannot be reduced to an absolute expression when it is encountered (see section **3.3.7.3 Expression Evaluation**), the displacement is sized according to the current set of displacement sizing options. These options are summarized in Table 3.14 and Table 3.15, which show the sizing rules for unknown absolute and PC-relative displacements, respectively. For more information on these options, see the **.opt / OPT** directive.

Size	Option (asm68)	Option (asm68k)
word	**a16**	**FRS**
long	**a32**	**FRL**†

† The displacement size is word if either the current section or the section in which the displacement is defined has the **.S** qualifier; the size is also word if the displacement is a symbol defined with the **COMM.S** or **LCOMM.S** directive, or declared external with the **XREF.S** directive.

Table 3.14: Unknown Absolute Displacement Sizing

Size	Option (asm68)	Option (asm68k)
byte	**fr8**	**BRB / BRS**
word	**fr16**	**BRW**

Table 3.15: Unknown PC-Relative Displacement Sizing

The options described above are overridden when either hole compression permits the use of a word displacement or the old-style displacement syntax is used. If a base and index register are used with the old-style syntax, the displacement is a byte; if only a base register is used, the displacement is a word.

3.5.4.3. Mode selection

After PC-relative coercion has been attempted and displacement sizing has been performed, the specified address consists of one or more of the following (see Table 3.11 for addressing modes supported by Texas Instruments):

- Base displacement (minimally sized).
- Outer displacement (minimally sized).
- Base register (possibly PC resulting from coercion).
- Index register.

If the address consists of a base displacement with no memory indirection, then the absolute short or absolute long addressing mode is selected depending on the size of the displacement.

If the address consists of a byte displacement, a base register, an index register, and no memory indirection, then the (**d_8,An,Xn**) or (**d_8,PC,Xn**) mode is selected. When assembling for the 68000, the displacement can be omitted provided that the **.opt iopt / OPT IOPT** directive is in effect (default).

If the address consists of a byte or word displacement, a base register, and no memory indirection, then the (**d_{16},An**) or (**d_{16},PC**) mode is selected. The

address (**0,An**) is optimized to (**An**) when the **.opt iopt / OPT IOPT** directive is in effect (default).

Otherwise, the appropriate register indirect or memory indirect mode is selected as determined by the syntax (i.e., the position of the square brackets or a lack thereof). The individual components of the address are determined as follows:

- If the base register has been omitted, then a suppressed address register is used. This condition implies that PC-relative coercion has failed.
- If the index register has been omitted, then a suppressed index register is used. If memory indirection is being performed, the preindexed mode is used (as opposed to the postindexed mode). The choice of mode, however, does not affect the effective address calculation.
- If the base displacement has been omitted, a null displacement is used. A word displacement is used in the case of a byte-size displacement.
- If the outer displacement has been omitted, a null displacement is used. A word displacement is used in the case of a byte-size displacement.

3.6. Asm68 Assembler Directives

Assembler directives are used in assembly source to generate data and to alter the assembler's object code generation behavior. This section describes the various directives supported by **asm68** and supplies examples of their typical usage. The directives are first presented in functional groups, a summary of which is presented in Table 3.16. For ease of reference, full descriptions of the directives, including syntax and examples, are then provided in an alphabetically arranged format.

Group	Description
Section directives	Create and resume sections
Symbol directives	Create and modify symbols
Data/Fill directives	Generate initialized/uninitialized data
Control directives	Control assembly
Output directives	Specify output settings
Debugging directives	Generate debugging information

Table 3.16: Directive Groups

The mnemonics for the **asm68** assembler directives, like those for instructions, are written in either all uppercase or all lowercase characters. As stated earlier, the mnemonics for **asm68** are shown in lowercase characters only to differentiate them from those for **asm68k**, which are shown in uppercase characters.

3.6.1. Asm68 Section Directives

Section directives can be used to manage both absolute and relocatable sections with any of the following types: text-type, data-type, or BSS-type. Text-type sections contain read-only data. Data-type sections contain initialized read/write data. BSS-type sections contain uninitialized read/write data. Section directives remain in effect until another section directive is issued.

Section directives can also be used to create structure template sections; these are special dummy sections that allow the convenient definition of labels suitable for structure field references. Table 3.17 summarizes the section directives for **asm68**. For more information, see section **3.3.4 Sections**. See chapter **7. Flash Application Layout** in the TI-89 / TI-92 Plus Developer Guide for information on the use and initialization of the TI-89 / TI-92 Plus sections.

Directive	Function
.bsection	Begin/resume a given BSS-type section
.bss	Begin/resume the BSS-type section **.bss**
.data	Begin/resume the data-type section **.data**
.dsection	Begin/resume a given data-type section
.ends	End a structure template section
.org	Begin an unnamed, absolute, data-type section
.reorg	Reset the location counter in an absolute section
.section	Begin/resume a given data-type section
.struct	Begin a structure template section
.text	Begin/resume the text-type section **.text**
.tsection	Begin/resume a given text-type section

Table 3.17: Section Directives

3.6.2. Asm68 Symbol Directives

Symbol directives are used to define symbols, declare their scopes, and set their values. By default, symbols have static scope, i.e., they are local to the files in which they are defined. In order to be referenced from within other files, they must be declared to have external scope. Table 3.18 summarizes the symbol directives for **asm68**. For more information, see section **3.3.5 Symbols**.

Directive	Function
.comm	Define a comm symbol
.extern	Declare external a referenced symbol
.external	Declare external a referenced symbol
.global	Declare external a defined symbol
.globl	Declare external a defined symbol
.lcomm	Define an lcomm symbol
.xdef	Declare external a defined symbol
.xref	Declare external a referenced symbol

Table 3.18: Symbol Directives

3.6.3. Asm68 Data/Fill Directives

Data directives are used to generate integer and floating-point data. Integer data can be expressed as integer constants, character constants, or any integer expression (see section **3.3.7 Expressions**). Floating-point data can be expressed as either floating-point symbols or floating-point constants. A single data directive can be used to generate multiple data items. Fill directives are used to allocate storage, typically for uninitialized data. Table 3.19 summarizes the data and fill directives for **asm68**. For more information, see sections **3.3.5 Symbols** and **3.3.6 Constants**.

Directive	Function
.align	Align location counter
.ascii	Generate integer data (byte)
.bin	Include contents of binary file
.byte	Generate integer data (byte)
.double	Generate TI BCD floating-point data
.extend	Not supported
.fill	Generate a block of initialized data
.float	Generate TI BCD floating-point data
.fpdata	Not supported
.long	Generate integer data (long-word)
.packed	Not supported
.short	Generate integer data (word)
.single	Not supported
.space	Generate a block of uninitialized data
.word	Generate integer data (word)

Table 3.19: Data/Fill Directives

3.6.4. Asm68 Control Directives

Assembly control directives provide mechanisms for controlling when and how instructions and directives are assembled. Their uses include option setting, conditional assembly, and source file inclusion. Table 3.20 summarizes the assembly control directives for **asm68**.

Directive	Function
.cmnt	Begin comment block
.elifdef	Assemble if alternative symbol defined
.else	Assemble if converse true
.end	End assembly
.endc	End comment block
.endif	End conditional assembly
.ifdef	Assemble if symbol defined
.ifndef	Assemble if symbol not defined
.include	Include assembler source file
.opt	Set assembler options

Table 3.20: Control Directives

3.6.5. Asm68 Output Directives

Output directives are used to format assembly listing files and to generate diagnostic messages. Table 3.21 summarizes the output directives for **asm68**.

Directive	Function
.echo	Echo message
.page	Begin new listing page

Table 3.21: Output Directives

3.6.6. Asm68 Debugging Directives

Debugging directives are used to generate source-level debugging information. They are typically generated by the Sierra Systems C compiler to allow source-level debugging, but can also be used when programming in assembly language. Table 3.22 summarizes the debugging directives for **asm68**.

The debugging directive descriptions are provided primarily to facilitate interpreting the compiler-generated directives. Their use is not recommended for programs written in assembly language. Instead, the **-L** command line flag should be used in conjunction with the **.type** directive to provide a reasonable level of debugging capability. When the **-L** command line flag is specified, a line number entry is generated for each instruction and memory allocation directive in text-type sections.

Directive	Function
.def	Begin symbol attribute block
.dim	Set array dimension attribute
.endef	End symbol attribute block
.file	Set name of source file
.line	Set line number attribute
.ln	Create line number entry
.scl	Set storage class attribute
.size	Set size attribute
.tag	Set tag name attribute
.type	Set type attribute
.val	Set value attribute

Table 3.22: Debugging Directives

3.6.7. Asm68 Directive Reference

The remainder of this section provides, in alphabetical order, detailed descriptions of the directives supported by **asm68**.

.align — Align Location Counter

Syntax

.align { 1 | 2 | 4 | 8 }

Description

The **.align** directive aligns the current section's location counter to the nearest multiple of the specified byte count. Any required padding is filled with the current fill value (see **.opt fillval**).

Example

```
.align 2
.word  0x4000
```

.ascii — Generate Integer Data (Byte)

Syntax

.ascii *operand* [, *operand*] . . .

Description

operand Specifies an integer expression.

The **.ascii** directive generates byte integer data. The values of the specified operands are placed in successive bytes beginning at the current location in the current section. This directive is a synonym for the **.byte** directive. For additional information, see section **3.3.6.2 Character Constants**.

Examples

```
.ascii "Hello!\0"                ; 4865 6c6c 6f21 00
.ascii 'a'                       ; 61
.ascii %1111,017,15,0xf          ; 0f0f 0f0f
.ascii 16 * 4 + 3                ; 43
```

Each of the above examples is shown with the code sequence (in hexadecimal) that it generates.

.bin — Include Contents of Binary File

Syntax

.bin "*filename*"

Description

filename The name of a binary file (including an optional absolute or relative path).

The **.bin** directive inserts the contents of the specified binary file at the current position in the assembler source. If the file is not specified with a full path, it is searched for in (or, relative to) the following directories in the indicated order:

1. The current directory.
2. Directories specified with the **`-I`** flag.
3. Directories specified with the environment variables **INCLUDE68** or **SIERRA** (see section **3.2.4 Environment Variables**).

Examples

```
.bin "table.inc"
.bin "../include/graphics.std"
```

.bsection — Begin / Resume a BSS-type Section

Syntax

.bsection *name* [, *address*]

Description

name Specifies the section. It is a symbol whose name can be up to eight characters in length. The first character of the name can be a period (.).

address Specifies the base or continuation address of the section. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.bsection** directive begins a BSS-type section with name *name*. If *address* is specified, the section is absolute and begins at that address; otherwise, the section is relocatable.

If the specified section already exists, it is resumed either at its current location (i.e., the value of its location counter) or at the specified address *address*. An absolute section can be restarted at any address beyond its current location. Relocatable sections cannot be resumed with an address specification.

No object code can be generated in BSS-type sections; they contain only uninitialized read/write data.

Examples

```
.bsection zero            ; relocatable
.bsection stats, 0x4000   ; absolute
```

.bss — Begin / Resume the BSS-type Section .bss

Syntax

.bss

Description

The **.bss** directive begins or resumes the BSS-type section **.bss**. It is functionally equivalent to **.bsection .bss**.

The section **.bss** contains only uninitialized data; therefore, no object code can be generated in this section. The section **.bss** is present in all object files, regardless of whether it is specified. For more information, see section **3.3.4.1 Section Types**.

.byte — Generate Integer Data (Byte)

Syntax

.byte *operand* [, *operand*] . . .

Description

operand Specifies an integer expression.

The **.byte** directive generates byte integer data. The values of the specified operands are placed in successive bytes beginning at the current location in the current section. This directive is a synonym for the **.ascii** directive. For additional information, see section **3.3.6.2 Character Constants**.

Examples

```
.byte "Hello!\0"                ; 4865 6c6c 6f21 00
.byte 'a'                       ; 61
.byte %1111,017,15,0xf          ; 0f0f 0f0f
.byte 16 * 4 + 3                ; 43
```

Each example is shown with the generated code sequence (in hexadecimal).

.cmnt — Begin Comment Block

Syntax

.cmnt

Description

The **.cmnt** directive begins a comment block. All assembler statements between this directive and its matching **.endc** directive are ignored. Pairs of comment block directives can be nested.

Example

```
.cmnt
    These lines are ignored by the assembler;
    no other comment markers are needed.
.endc
```

.comm — Define a comm Symbol

Syntax

.comm *symbol*, *count* [, *align*]

Description

symbol Specifies a symbol that is not defined elsewhere in the current file.

count Specifies the number of bytes associated with the symbol. It is an absolute expression that cannot contain any forward, external, or undefined references.

align Specifies the alignment requirements for the symbol. Its value can be 1, 2, or 4. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.comm** directive defines the specified symbol *symbol* and associates with it a block of uninitialized data in the BSS-type section **.bss**. The number of bytes in this block is specified by *count*. The alignment of the block is specified by *align*; if omitted or if the **-c** flag has been specified, quad alignment is used.

The scope of the symbol *symbol* is external. The block of data is allocated during linkage, unless the **-6** flag has been specified, in which case it is allocated during assembly. For more information, refer to section **3.3.5.4 Comm and Lcomm Symbols**.

Examples

```
.comm table, 4096
.comm strings, 256, 2
```

.data — Begin / Resume the Data-type Section .data

Syntax

.data

Description

The **.data** directive begins or resumes the data-type section **.data**. It is functionally equivalent to **.dsection .data**.

The section **.data** contains initialized read/write data. It is present in all object files, regardless of whether it is specified. For additional information, refer to section **3.3.4.1 Section Types**.

.def — Begin Symbol Attribute Block

Syntax

.def *symbol*

Description

symbol Specifies a symbol that is defined in the current assembler source file or corresponding C source file.

The **.def** directive begins an attribute block for the symbol *symbol*. The **.dim**, **.line**, **.scl**, **.size**, **.tag**, **.type**, and **.val** directives are used to set the various attributes of a symbol (see section **1.4 Object File Format**). The **.endef** directive must be used to end an attribute block. For convenience, the directives that comprise a symbol attribute block can be specified as a backslash-separated list.

The information contained in an attribute block is stored in the object file's symbol table for purposes of symbolic debugging. The Sierra C compiler automatically generates these attribute blocks for all symbols when the **-q** command line flag is specified. They can be written manually when performing assembler source-level debugging, but this is not recommended. Adequate debugging information can be generated with the **-L** assembler command line flag, which directs the assembler to generate line number information, and with the **.type** directive, which can be used to specify symbol types directly (i.e., without a symbol attribute block).

Examples

```
.def init
.val init
.scl 2
.type 0x24
.endef

.def tbl \ .val 12 \ .scl 3 \ .type 0x4 \ .endef
```

.dim — Set Array Dimension Attribute

Syntax

.dim ***dim*** **[,** ***dim*** **[,** ***dim*** **[,** ***dim*****]]]**

Description

dim Specifies an array dimension of the current symbol. It is an absolute expression that cannot contain any forward, external, or undefined references. Up to four dimensions can be specified.

The **.dim** directive sets the dimension attribute of the symbol referenced by the current attribute block (see **.def** directive). The dimension attribute is specified for array types (see sections **1.4.8.7 Type Entry** and **1.4.9.6 Arrays**). This directive can appear at most once per symbol attribute block.

The **.dim** directive is typically used only for C source-level debugging; it is ignored when assembler source-level debugging information is generated with the **-L** command line flag (see also **.type** directive).

Example

```
.def buf
.val buf
.dim 16,4           ; buf is a two-dim array, int buf[16][4]
.scl 2
.type 0xf4
.line 25
.size 256
.endef
```

.double — Generate Floating-Point Data

Syntax

.double ***operand*** **[,** ***operand*****] . . .**

Description

operand Specifies a floating-point symbol or floating-point constant. No forward references are allowed.

The **.double** directive generates TI BCD floating-point data. The values of the specified operands are placed in 10 bytes beginning at the current location in the current section. A warning is issued if the alignment is odd.

Examples

```
_PI:
   .double 3.141592653589793        ; 4000 3141 5926 5358 9793
   .double 0X40003141592653589793   ; 4000 3141 5926 5358 9793
   .double -10,.2                   ; c001 1000 0000 0000 0000
                                    ; 3fff 2000 0000 0000 0000
```

Each of the above examples is shown with the sequence of words (in hexadecimal) that it generates.

.dsection — Begin / Resume a Data-type Section

Syntax

.dsection *name* [, *address*]

Description

name Specifies the section. It is a symbol whose name can be up to eight characters in length. The first character of the name can be a period (.).

address Specifies the base or continuation address of the section. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.dsection** directive begins a data-type section with name *name*. If *address* is specified, the section is absolute and begins at that address; otherwise, the section is relocatable.

If the specified section already exists, it is resumed either at its current location (i.e., the value of its location counter) or at the specified address *address*. An absolute section can be restarted at any address beyond its current location; any space that is created is filled with the current fill value (see **.opt fillval**). Relocatable sections cannot be resumed with an address specification.

Examples

```
.dsection table           ; relocatable
.dsection ram, 0x8000     ; absolute
```

.echo — Echo Message

Syntax

.echo { *text* | *expression*[*.format*] } . . .

Description

text Specifies an ASCII string. Whitespace characters are permitted, but curly braces ({ }) are not.

expression Specifies an absolute expression that cannot contain any forward, external, or undefined references. The expression must be enclosed in curly braces ({ });

format Specifies the format for printing the preceding expression. The legal format characters, which correspond to those of the ANSI standard C library function **printf**, are shown below; the default is **d**.

- **d** Signed decimal
- **u** Unsigned decimal
- **o** Octal
- **x** Hexadecimal (lower case)
- **X** Hexadecimal (upper case)

The **.echo** directive causes the specified message to be written to the standard output stream, **stdout**. The end of the message is marked by the newline character; therefore, no comment field is permitted with this directive. This directive can be used for both informational and diagnostic purposes.

Example

```
.echo The end of the data section is {data_end}.x
```

The symbol data_end is a user-defined label, which has been defined prior to the directive in this example. If the value of the label is 0x42f8, then this directive produces the following message:

```
The end of the data section is 42f8
```

.elifdef — Assemble If Alternative Symbol Defined

Syntax

.elifdef *symbol*

Description

symbol Specifies a user-defined symbol.

The **.elifdef** directive is used in a conditional assembly block. It is equivalent to the **.else** directive followed by an **.ifdef – .endif** block (see **.ifdef** and **.endif** directives). Its use obviates the need for multiple **.endif** directives in conditional blocks that have multiple alternatives.

The **.elifdef** directive is optional within a conditional assembly block.

Example

```
.ifdef option1
    moveq        #1,d0
.elifdef option2
    moveq        #2,d0
.elifdef option3
    moveq        #3,d0
.endif
```

.else — Assemble If Converse True

Syntax

.else

Description

The **.else** directive is used in a conditional assembly block. This directive matches the immediately preceding **.ifdef**, **.ifndef**, or **.elifdef** directive that is not matched by a **.endif** or **.else** directive. If this preceding directive fails, then the statements between the **.else** directive and the matching **.endif** directive are assembled; otherwise, these statements are skipped.

The **.else** directive is optional within a conditional assembly block.

Example

```
.ifdef debug
    moveq        #1,d0
.else
    moveq        #0,d0
.endif
```

.end — End Assembly

Syntax

.end

Description

The **.end** directive ends assembly in the current source file. Any assembler statements appearing after this directive are ignored.

.endc — End Comment Block

Syntax

.endc

Description

The **.endc** directive ends a comment block (see **.cmnt** directive).

.endef — End Symbol Attribute Block

Syntax

.endef

Description

The **.endef** directive ends the current symbol attribute block (see **.def** directive).

.endif — End Conditional Assembly Block

Syntax

.endif

Description

The **.endif** directive ends a conditional assembly block (see **.ifdef** and **.ifndef** directives).

Example

```
.ifdef   debug
    jsr  mem_dump
.endif
```

.ends — End a Structure Template Section

Syntax

.ends

Description

The **.ends** directive ends a structure template section. The section that was active prior to the structure template section is automatically resumed. Refer to the description of the **.struct** directive for more information.

.extend — Generate Floating-Point Data (Extended-Precision)

Not supported by Texas Instruments. However, **.extend** is still recognized as a reserved name by **asm68**.

.extern / .external — Declare External a Referenced Symbol

Syntax

.extern *symbol* [, *symbol*] . . .

.external *symbol* [, *symbol*] . . .

Description

symbol Specifies a symbol that is referenced (but not defined) in the current file. Section names and floating-point symbols are not allowed.

The **.extern** and **.external** directives declare the scope of the symbol *symbol* to be external. This is necessary when a locally referenced symbol is defined in another source file. These directives are synonyms for the **.xref** directive.

Symbols that are referenced in a file but not defined in that file are assumed to have external scope; therefore, these directives are not necessary (unless the **-E** flag is specified on the command line).

Examples

```
.extern   base, init, input
.external proc1, jmp_tbl
```

.file — Set Name of Source File

Syntax

.file "*filename*"

Description

filename Specifies the name of either the assembler source file or the corresponding C source file. It can be up to 14 characters in length.

The **.file** directive sets the name of the source file for purposes of source-level debugging. This directive can appear at most once per source file. It is generated internally by the assembler if it does not appear or if it specifies a C source file when the **-L** command line flag is used. For more information, see section **1.4.7.1 Special Symbols**.

Example

```
.file  "demo.c"
```

.fill — Allocate a Block of Initialized Memory

Syntax

.fill[*.size*] *count*, *value*

Description

size Specifies the unit size. The legal size qualifiers are shown below; the default is **b**.

b	Byte Integer (1 byte)
w	Word Integer (2 bytes)
l	Long-word Integer (4 bytes)
s	Single-precision Real (not supported)
d	Double-precision Real (not supported)
x	Extended-precision Real (not supported)
p	Packed Decimal Real (not supported)

count Specifies the unit count. It is an absolute expression that cannot contain any forward, external, or undefined references.

value Specifies the unit value. If *size* is **w** or **l**, any integer expression can be used; otherwise, an absolute expression that contains no forward, external, or undefined references must be used.

The **.fill** directive allocates a block of initialized memory whose size is determined by the number and size of the unit location. Each location is filled with the specified value *value*. This directive cannot be used in BSS-type sections.

Examples

```
.fill   256, 0xff
.fill.l 16, err_vector
```

.float — Generate Floating-Point Data (Single-Precision)

Syntax

.float ***operand*** **[,** ***operand*****] . . .**

Description

operand Specifies a floating-point symbol or floating-point constant. No forward references are allowed.

The **.float** directive generates TI BCD floating-point data. The values of the specified operands are placed in 10 bytes beginning at the current location in the current section. A warning is issued if the alignment is odd. The data generated is the same as the **.double** directive, allowing 16 digits in the mantissa. Since a float in the compiler, **com68**, contains only 14 significant digits in the mantissa, it is recommended to always use **.double** to avoid confusion.

Examples

```
.float 3.141592653589793   ; 4000 3141 5926 5358 9793
.float -10,.2              ; c001 1000 0000 0000 0000
                           ; 3fff 2000 0000 0000 0000
```

Each of the above examples is shown with the sequence of words (in hexadecimal) that it generates.

.fpdata — Generate Floating-Point Data

Not supported by Texas Instruments; however, **.fpdata** is still recognized as a reserved name by **asm68**.

.global / .globl — Declare External a Defined Symbol

Syntax

.global *symbol* [, *symbol*] . . .

.globl *symbol* [, *symbol*] . . .

Description

symbol Specifies a symbol that is defined in the current file. Section names and floating-point symbols are not allowed.

The **.global** and **.globl** directives declare the scope of the symbol *symbol* to be external. This is necessary when the symbol is referenced in other source files since the default symbol scope is static. These directives are synonyms for the **.xdef** directive.

Examples

```
.globl   main, jmp_tbl
.global  diag_list, proc1, eval
```

.ifdef — Assemble If Symbol Defined

Syntax

.ifdef *symbol*

Description

symbol Specifies a user-defined symbol.

The **.ifdef** directive introduces a conditional assembly block. If the specified symbol *symbol* is defined when the directive is encountered, then the statements between this directive and the first matching **.elifdef**, **.else**, or **.endif** directive are assembled and the remainder of the block is skipped. Otherwise, the statements associated with the **.ifdef** directive are skipped and control passes to the aforementioned matching directive.

Conditional assembly directives can be nested to 40 levels.

Example

```
.ifdef serial
    move.l  #serial_dev, io_func
.endif
```

.ifndef — Assemble If Symbol Not Defined

Syntax

.ifndef *symbol*

Description

symbol Specifies a user-defined symbol.

The **.ifndef** directive introduces a conditional assembly block. If the specified symbol *symbol* is not defined when the directive is encountered, then the statements between this directive and the first matching **.elifdef**, **.else**, or **.endif** directive are assembled and the remainder of the block is skipped. Otherwise, the statements associated with the **.ifdef** directive are skipped and control passes to the aforementioned matching directive.

Conditional assembly directives can be nested to 40 levels.

Example

```
.ifndef serial
    move.l  #parallel_dev, io_func
.endif
```

.include — Include Assembler Source File

Syntax

.include "*filename*"

Description

filename The name of an assembler source file (including an optional absolute or relative path).

The **.include** directive inserts the contents of the specified file at the current position in the assembler source. If the file is not specified with a full path, it is searched for in (or, relative to) the following directories in the indicated order:

1. The current directory.
2. Directories specified with the **`-I`** flag.
3. Directories specified with the environment variables **INCLUDE68** or **SIERRA** (see section **3.2.4 Environment Variables**).

This directive can be nested, i.e., included files can themselves include files. The assembler imposes no limit on the level of nesting.

Examples

```
.include "table.inc"
.include "../include/vector.h"
```

.lcomm — Define an lcomm Symbol

Syntax

.lcomm *symbol*, *count* [, *align*]

Description

symbol Specifies a symbol that is not defined elsewhere in the current file.

count Specifies the number of bytes associated with the symbol. It is an absolute expression that cannot contain any forward, external, or undefined references.

align Specifies the alignment requirements for the symbol. Its value can be 1, 2, or 4. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.lcomm** directive defines the specified symbol *symbol* and associates with it a block of uninitialized data in the BSS-type section **.bss**. The number of bytes in this block is specified by *count*. The alignment of the block is specified by *align*; if omitted or if the **-c** flag has been specified, quad alignment is used.

The scope of the symbol *symbol* is static. The block of data is allocated during assembly. For more information, refer to section **3.3.5.4 Comm and Lcomm Symbols**.

Examples

```
.lcomm table, 4096
.lcomm strings, 256, 4
```

.line — Set Line Number Attribute

Syntax

.line ***line***

Description

line Specifies the number of the line on which the current symbol is defined. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.line** directive sets the line number attribute of the symbol referenced by the current attribute block (see **.def** directive). For a detailed description of this attribute, see section **1.4.9 Auxiliary Table Entries**. This directive can appear at most once per symbol attribute block.

The **.line** directive is typically used only for C source-level debugging; it is ignored when assembler source-level debugging information is generated with the **-L** command line flag (see also **.type** directive).

Example

```
.def buf
.val buf
.dim 16,4
.scl 2
.type 0xf4
.line 25                    ; buf is defined on line 25
.size 256
.endef
```

.ln — Create Line Number Entry

Syntax

.ln ***line***

Description

line Specifies the current line number in the source file. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.ln** directive creates a line number entry for purposes of source-level debugging. This entry associates the current section's location counter with a line in the associated C source file. For more information, refer to section **1.4.6 Line Number Information**. This directive cannot be used in BSS-type sections.

The **.ln** directive is typically used only for C source-level debugging; it is ignored when assembler source-level line number information is generated with the **-L** command line flag.

Example

```
.ln 33
add d0,d1
```

.long — Generate Integer Data (Long-Word)

Syntax

.long ***operand*** **[,** ***operand*****] . . .**

Description

operand Specifies an integer expression.

The **.long** directive generates long-word integer data. The values of the specified operands are placed in successive long words beginning at the current location in the current section. A warning is issued if the alignment is odd. For additional information, see section **3.3.6.2 Character Constants**.

Examples

```
.long 'abcd'                    ; 61626364
.long 16777215, 0xffffff        ; 00ffffff 00ffffff
.long 0xff << 24                ; ff000000
```

Each of the above examples is shown with the sequence of long words (in hexadecimal) that it generates.

.opt — Set Assembler Options

Syntax

.opt ***option*** **[,** ***option*****] . . .**

Description

option An assembler option.

The **.opt** directive sets the specified assembler options. These options affect the assembly of instructions, effective addresses, and data. They also provide a means of customizing the assembly listing output. Only options valid for the 68000 are supported by Texas Instruments.

Options

a16 **a32**	Set the size of unknown absolute displacements to either 16 or 32 bits (see section **3.5.4.2 Displacement Sizing**). (Default: **a32**)
dlist **nodlist**	Enable/disable full listing of data directive assembly. The **dlist** option allows the object code associated with data directives (e.g., **.byte**, **.word**, etc.) to be listed in its entirety, while the **nodlist** option permits only one line of object code to be listed. (Default: **dlist**)
eqin **noeqin**	Enable/disable inclusion of assignment symbols (i.e., symbols created by direct assignment) in the object file's symbol table. (Default: **eqin**)
ffp **noffp**	Enable/disable Motorola fast floating-point format conversion (not supported by Texas Instruments).
fillval=*value*	Set the fill value to *value* (see **.align** and **.space** directives). A warning is issued if the specified value does not fit in a signed or unsigned byte. (Default: **fillval=0**)
fpid=*n*	Set the identification number of the 68881/2 coprocessor (not supported by Texas Instruments).
fpisize=*size*	Set the default floating-point instruction size (not supported by Texas Instruments).
fr8 **fr16**	Set the size of unknown PC-relative displacements to 8 or 16 bits (see section **3.5.4.2 Displacement Sizing**). (Default: **fr16**)

fwdsize Enable displacement size checking for forward branches. The assembler will issue a warning for each displacement that can be reduced in size.
(Default: **disabled**)

iopt
noiopt Enable/disable instruction optimizations (see section **3.4.3 Instruction Optimization**).
(Default: **iopt**)

isize=*size* Set the default instruction size (see section **3.4.2 Instruction Sizing**). The following are the legal sizes:

b Byte Integer

w Word Integer

l Long-word Integer

(Default: **isize=l**)

lhex List the alphabetic hexadecimal digits using lowercase characters (see also **uhex**).
(Default: **lhex**)

list
nolist Enable/disable listing of the assembly code. These options are used to omit portions of the assembly code from the listing file. Pairs of these complement options can be nested.
(Default: **list**)

nopc Disable all coercions to PC-relative addressing modes. This option disables the **pcb16**, **pcb32**, and **pcf** options.
(Default: **pcb16**)

pca
nopca Enable/disable coercion to PC-relative addressing modes for references to an absolute location or from an absolute section. The **pcf** option must be enabled for this option to be effective (see section **3.5.4.1 PC-relative Coercion**).
(Default: **pca**)

pcb16
pcb32 Enable coercion to PC-relative addressing modes for 16-bit and 32-bit backward references (see section **3.5.4.1 PC-relative Coercion**). Selecting the **pcb16** option disables the **pcb32** option; selecting the **pcb32** option enables the **pcb16** option. The **pcb32** option is not legal on the 68000/10.
(Default: **pcb16**)

pcf Enable coercion to PC-relative addressing modes for forward references, references to locations positioned at an unknown distances. This option also enables the **pcb16** option. For more information, see section **3.5.4.1 PC-relative Coercion**. (Default: **pcb16**)

prec=*type* Set the precision used when converting floating-point values (not supported by Texas Instruments).

proc=*proc* Set the target processor. **68000** is the only processor recognized by Texas Instruments. (Default: **proc=68000**)

rngchk=*type* Set the immediate data range check for byte and word data. The following four levels of range checking are provided:

- **0** No checking: all values are silently truncated
- **1** Byte: -256–255; Word: -65536–65535
- **2** Byte: -128–255; Word: -32768–65535
- **3** Byte: -128–127; Word: -32768–32767

(Default: **rngchk=1**)

round=*mode* Set the rounding mode used when converting floating-point values (not supported by Texas Instruments).

title *title* Set the title that appears at the top of each page of the assembly listing. The string *title* must be delimited by a character that does not appear in the title itself (e.g., double quotes). Ending the title with a tilde (~) omits the source file name from the listing header. (Default: title **"Sierra Systems ASM68 x.xx"**)

uhex List the alphabetic hexadecimal digits using uppercase characters. (Default: **lhex**)

uloc Modify the naming convention for compiler locals by adding an initial underscore (_). For more information, see section **3.3.5.6 Compiler Locals**. (Default: **disabled**)

.org — Begin an Absolute Data-type Section

Syntax

.org ***address***

Description

address Specifies the base address of the section. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.org** directive begins an unnamed, data-type section. The section's base address *address* is fixed.

Examples

```
.org 0x8000
.org base + 1024
```

In the second example, the symbol **base** is absolute and is defined prior to this directive.

.packed — Generate Floating-Point Data (Packed Decimal)

Not supported by Texas Instruments. However, **.packed** is still recognized as a reserved name by **asm68**.

.page — Begin New Listing Page

Syntax

.page

Description

The **.page** directive begins a new page in the listing file. This directive does not appear in the listing.

.reorg — Reset the Location Counter in an Absolute Section

Syntax

.reorg *address*

Description

address Specifies the address at which the current section will continue. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.reorg** directive resets the location counter in an absolute section. The continuation address *address* must be greater than the current value of the section's location counter; any space that is created is filled with the current fill value (see **.opt fillval**), unless the section is of BSS-type.

Examples

```
.org 0x4000
      addq      #7,d0
.reorg 0x6000
```

```
.bsection tbl, 0x1000
      move.l    _abc,d0
.reorg 0x8000
```

As the second example illustrates, the **.reorg** directive can be applied to any absolute section, regardless of its declaration.

.scl — Set Storage Class Attribute

Syntax

.scl ***class***

Description

class Specifies the storage class of the current symbol. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.scl** directive sets the storage class attribute of the symbol referenced by the current attribute block (see **.def** directive). For a list of recognized storage classes, see section **1.4.8.2 Storage Class**. This directive can appear at most once per symbol attribute block.

The **.scl** directive is typically used only for C source-level debugging; it is ignored when assembler source-level debugging information is generated with the **-L** command line flag (see also **.type** directive).

Example

```
.def x
.val 3
.scl 4              ; x is in a register
.type 4
.endef
```

.section — Begin / Resume a Data-type Section

Syntax

.section *name* [, *address*]

Description

name Specifies the section. It is a symbol whose name can be up to eight characters in length. The first character of the name can be a period (.).

address Specifies the base or continuation address of the section. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.section** directive begins a data-type section with name *name*. If *address* is specified, the section is absolute and begins at that address; otherwise, the section is relocatable.

If the specified section already exists, it is resumed either at its current location (i.e., the value of its location counter) or at the specified address *address*. An absolute section can be restarted at any address beyond its current location; any space that is created is filled with the current fill value (see **.opt fillval**). Relocatable sections cannot be resumed with an address specification.

This directive is a synonym for the **.dsection** directive.

Examples

```
.section table                  ; relocatable
.section ram, 0x8000            ; absolute
```

.short — Generate Integer Data (Word)

Syntax

.short *operand* [, *operand*] . . .

Description

operand Specifies an integer expression.

The **.short** directive generates word integer data. The values of the specified operands are placed in successive words beginning at the current location in the current section. A warning is issued if the alignment is odd. This directive is a synonym for the **.word** directive. For additional information, refer to section **3.3.6.2 Character Constants**.

Examples

```
.short 'ab'                    ; 6162
.short 01777,1023,0x3ff        ; 03ff 03ff 03ff
.short 1024 * 16               ; 4000
```

Each of the above examples is shown with the sequence of words (in hexadecimal) that it generates.

.single — Generate Floating-Point Data (Single-Precision)

Not supported by Texas Instruments. However, **.single** is still recognized as a reserved name by **asm68**.

.size — Set Size Attribute

Syntax

.size ***size***

Description

size Specifies the size of the current symbol. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.size** directive sets the size attribute of the symbol referenced by the current attribute block (see **.def** directive). The size attribute is specified for aggregate types (see section **1.4.9 Auxiliary Table Entries**). This directive can appear at most once per symbol attribute block.

The **.size** directive is typically used only for C source-level debugging; it is ignored when assembler source-level debugging information is generated with the **-L** command line flag (see also **.type** directive).

Example

```
.def buf
.val buf
.dim 16,4
.scl 2
.type 0xf4
.line 25
.size 256           ; buf is an array 256 bytes long
.endef
```

.space — Allocate a Block of Uninitialized Memory

Syntax

.space[.*size*] *count*

Description

size Specifies the unit size. The legal size qualifiers are shown below; the default is **b**.

- **b** Byte Integer (1 byte)
- **w** Word Integer (2 bytes)
- **l** Long-word Integer (4 bytes)
- **s** Single-precision Real (not supported)
- **d** Double-precision Real (not supported)
- **x** Extended-precision Real (not supported)
- **p** Packed Decimal Real (not supported)

count Specifies the unit count. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.space** directive allocates a block of uninitialized memory whose size is determined by the number and size of the unit location. Each location is filled with the current fill value (see **.opt fillval**), unless the section is of BSS-type.

Examples

```
.space    256
.space.l  64
```

.struct — Begin a Structure Template Section

Syntax

.struct [*label*]

Description

label Specifies a label that is used for documentation purposes. It is not entered in the symbol table.

The **.struct** directive begins a structure template section. This type of section is used in conjunction with the **.space** and **.align** directives to define labels suitable for structure field references. These labels are not included in the object file's symbol table.

Structure template sections begin at absolute address zero. Since they are dummy sections, they cannot contain any object code. Also, they cannot be nested; however, fields of a nested structure can be treated as part of the enclosing structure. The **.ends** directive is used to end a structure template section and resume the previously active section.

Example

```
          .struct node                    ; struct node {
visited:  .space 1                        ; char visited;
          .align 2                        ; struct position {
pos.x:    .space 2                        ; short int x;
pos.y:    .space 2                        ; short int y;
          .align 4                        ; } pos;
left:     .space 4                        ; struct node *left;
right:    .space 4                        ; struct node *right;
          .ends                           ; };
          move.b        #1,_n+visited     ; n.visited = 1;
          movea.l       _p,a0             ; p–>right = NULL;
          clr.l         (right,a0)        ;
```

This example illustrates how the **.struct** and **.ends** directives are used to define a set of structure field labels. The **.space** and **.align** directives are used to allocate space and maintain proper alignment, respectively.

.tag — Set Tag Name Attribute

Syntax

.tag *symbol*

Description

symbol Specifies the tag name with which the current symbol is associated. It is a symbol that is defined as a structure, union, or enumeration type.

The **.tag** directive sets the tag name attribute of the symbol referenced by the current attribute block (see **.def** directive). For a detailed description of the tag name attribute, see section **1.4.9 Auxiliary Table Entries**. This directive can appear at most once per symbol attribute block.

The **.tag** directive is typically used only for C source-level debugging; it is ignored when assembler source-level debugging information is generated with the **-L** command line flag (see also **.type** directive).

Example

```
.def x
.val x
.scl 2
.type 8
.size 4
.tag s              ; s is the tag of the structure
.endef              ; of which x is a member
```

.text — Begin / Resume the Text-type Section .text

Syntax

.text

Description

The **.text** directive begins or resumes the text-type section **.text**. It is functionally equivalent to **.tsection .text**.

The section **.text** contains read-only data. It is present in all object files, regardless of whether it is specified. For more information, see section **3.3.4.1 Section Types**.

.tsection — Begin/Resume a Text-type Section

Syntax

.tsection *name* [, *address*]

Description

name Specifies the section. It is a symbol whose name can be up to eight characters in length. The first character of the name can be a period (.).

address Specifies the base or continuation address of the section. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.tsection** directive begins a text-type section with name *name*. If *address* is specified, the section is absolute and begins at that address; otherwise, the section is relocatable.

If the specified section already exists, it is resumed either at its current location (i.e., the value of its location counter) or at the specified address *address*. An absolute section can be restarted at any address beyond its current location; any space that is created is filled with the current fill value (see **.opt fillval**). Relocatable sections cannot be resumed with an address specification.

Examples

```
.tsection code                  ; relocatable
.tsection rom, 0xf800           ; absolute
```

.type — Set Type Attribute

Syntax

.type ***type***

Description

type Specifies the type of the current symbol. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.type** directive sets the type attribute of the symbol referenced by the current attribute block (see **.def** directive). For a discussion of fundamental and derived types, see section **1.4.8.7 Type Entry**. This directive can appear at most once per symbol attribute block.

The **.type** directive can also be used in conjunction with the **-L** command line flag to provide type information for symbols. When the **-L** command line flag is used, this directive does not have to appear within a symbol attribute block; the type it specifies is associated with the most recently defined label. Function types can only be attributed to labels defined in text-type sections.

Example

```
x:
.type 0x24          ; x is a function that returns an int
```

.val — Set Value Attribute

Syntax

.val ***value***

Description

value Specifies the value of the current symbol. It is an absolute or simple relocatable expression.

The **.val** directive sets the value attribute of the symbol referenced by the current attribute block (see **.def** directive). The meaning of the value attribute is dependent upon the selected storage class (see section **1.4.8.4 Symbol Value Field**). This directive can appear at most once per symbol attribute block.

The **.val** directive is typically used only for C source-level debugging; it is ignored when assembler source-level debugging information is generated with the **-L** command line flag (see also **.type** directive).

Example

```
.def x
.val x          ; x is a variable — its value is its location
.scl 2
.type 4
.endef
```

.word — Generate Integer Data (Word)

Syntax

.word *operand* [, *operand*] . . .

Description

operand Specifies an integer expression.

The **.word** directive generates word integer data. The values of the specified operands are placed in successive words beginning at the current location in the current section. A warning is issued if the alignment is odd. For additional information, refer to section **3.3.6.2 Character Constants**.

Examples

```
.word 'ab'                          ; 6162
.word %1111111111,01777,1023        ; 03ff 03ff 03ff
.word 1024 * 16                     ; 4000
```

Each of the above examples is shown with the sequence of words (in hexadecimal) that it generates.

.xdef — Declare External a Defined Symbol

Syntax

.xdef *symbol* [, *symbol*] . . .

Description

symbol Specifies a symbol that is defined in the current file. Section names and floating-point symbols are not allowed.

The **.xdef** directive declares the scope of the symbol *symbol* to be external. This is necessary when the symbol is referenced in other source files (since the default symbol scope is static).

Examples

```
.xdef init
.xdef proc1, diag_list
```

.xref — Declare External a Referenced Symbol

Syntax

.xref *symbol* [, *symbol*] . . .

Description

symbol Specifies a symbol that is referenced (but not defined) in the current file. Section names and floating-point symbols are not allowed.

The **.xref** directive declares the scope of the symbol *symbol* to be external. This is necessary when a locally referenced symbol is defined in another source file.

Symbols that are referenced in a file but not defined in that file are assumed to have external scope; therefore, this directive is not necessary (unless the **-E** flag is specified on the command line).

Examples

```
.xref base, init, input
.xref proc1
```

3.7. Asm68k Assembler Directives

Assembler directives are used in assembly source to generate data and to alter the assembler's object code generation behavior. This section describes the various directives supported by **asm68k** and supplies examples of their typical usage. The directives are first presented in functional groups, a summary of which is presented in Table 3.23. For ease of reference, full descriptions of the directives, including syntax and examples, are then provided in an alphabetically arranged format.

Group	Description
Section directives	Create and resume sections
Symbol directives	Create and modify symbols
Data/Fill directives	Generate initialized/uninitialized data
Control directives	Control assembly
Output directives	Specify output settings
Debugging directives	Generate debugging information

Table 3.23: Directive Groups

The mnemonics for the **asm68k** assembler directives, like those for instructions, are written in either all uppercase or all lowercase characters. The lowercase mnemonics all begin with a period (.) and they are extensions to the Motorola M68000 Resident Structured Assembler. The additional mnemonics (from **asm68**) are needed to allow **asm68k** to support the assembly code generated by the Sierra Systems compiler, **com68**. Note that all mnemonics, regardless of the case they are shown in, can be written in either lowercase or uppercase characters. When two mnemonics are equivalent and referenced from within this document, the names of the mnemonics are shown separated by a slash (e.g., **.global / XDEF**).

3.7.1. Asm68k Section Directives

Section directives can be used to manage both absolute and relocatable sections with any of the following types: text-type, data-type, or BSS-type. Text-type sections contain read-only data. Data-type sections contain initialized read/write data. BSS-type sections contain uninitialized read/write data. Section directives remain in effect until another section directive is issued.

Section directives can also be used to create structure template sections; these are special dummy sections that allow the convenient definition of labels suitable for structure field references. Table 3.24 summarizes the section directives for **asm68k**. For more information, see section **3.3.4 Sections**. See chapter **7. Flash Application Layout** in the TI-89 / TI-92 Plus Developer Guide for information on the use and initialization of the TI-89 / TI-92 Plus sections.

Directive	Function
.bsection	Begin/resume a given BSS-type section
.bss	Begin/resume the BSS-type section **.bss**
.data	Begin/resume the data-type section **.data**
.dsection	Begin/resume a given data-type section
.text	Begin/resume the text-type section **.text**
.tsection	Begin/resume a given text-type section
BSECTION	Begin/resume a given BSS-type section
DSECTION	Begin/resume a given data-type section
OFFSET	Begin a structure template section
ORG	Begin an unnamed, absolute, data-type section
REORG	Reset the location counter in an absolute section
SECTION	Begin/resume a given data-type section
TSECTION	Begin/resume a given text-type section

Table 3.24: Section Directives

3.7.2. Asm68k Symbol Directives

Symbol directives are used to define symbols, declare their scopes, and set their values. By default, symbols have static scope, i.e., they are local to the files in which they are defined. In order to be referenced from within other files, they must be declared to have external scope. Table 3.25 summarizes the symbol directives for **asm68k**. For more information, see section **3.3.5 Symbols**.

Directive	Function
.comm	Define a comm symbol
.global	Declare external a defined symbol
.globl	Declare external a defined symbol
.lcomm	Define an lcomm symbol
COMM	Define a comm symbol
EQU	Define an integer symbol
FEQU	Not supported
LCOMM	Define an lcomm symbol
REG	Define a register list symbol
SET	Define/redefine an integer symbol
XDEF	Declare external a defined symbol
XREF	Declare external a referenced symbol

Table 3.25: Symbol Directives

3.7.3. Asm68k Data/Fill Directives

Data directives are used to generate integer and floating-point data. Integer data can be expressed as integer constants, character constants, or any integer expression (see section **3.3.7 Expressions**). Floating-point data can be expressed as either floating-point symbols or floating-point constants. A single data directive can be used to generate multiple data items. Fill directives are used to allocate storage, typically for uninitialized data. Table 3.26 summarizes the data and fill directives for **asm68k**. For more information, see sections **3.3.5 Symbols** and **3.3.6 Constants**.

Directive	Function
.align	Align location counter
.bin	Include contents of binary file
.byte	Generate integer data (byte)
.double	Generate TI BCD floating-point data
.extend	Not supported
.float	Generate TI BCD floating-point data
.long	Generate integer data (long-word)
.short	Generate integer data (word)
.space	Generate a block of uninitialized data
.word	Generate integer data (word)
BIN	Include contents of binary file
COMLINE	Generate a block of uninitialized data
DC	Generate integer/floating-point data
DCB	Generate a block of initialized data
DS	Generate a block of uninitialized data

Table 3.26: Data/Fill Directives

3.7.4. Asm68k Control Directives

Assembly control directives provide mechanisms for controlling when and how instructions and directives are assembled. Their uses include option setting, conditional assembly, and source file inclusion. Table 3.27 summarizes the assembly control directives for **asm68k**.

Directive	Function
.cmnt	Begin comment block
.elifdef	Assemble if alternative symbol defined
.else	Assemble if converse true
.endc	End comment block
.endif	End conditional assembly
.ifdef	Assemble if symbol defined
.ifndef	Assemble if symbol not defined
.include	Include assembler source file
.opt	Set assembler options
ELSEC	Assemble if converse true
END	End assembly
ENDC	End conditional assembly block
FOPT	Not supported
IFC	Assemble if strings equal
IFcc	Assemble if condition true
IFNC	Assemble if strings not equal
INCLUDE	Include assembler source file
MASK2	Not supported
OPT	Set assembler options

Table 3.27: Control Directives

3.7.5. Asm68k Output Directives

Output directives are used to format assembly listing files and to generate diagnostic messages. Table 3.28 summarizes the output directives for **asm68k**.

Directive	Function
.echo	Echo message
FAIL	Generate error message
FORMAT†	Format listing file
LIST	Enable assembly listing
LLEN	Set line length of listing file
NOFORMAT†	Do not format listing file
NOL	Disable assembly listing
NOLIST	Disable assembly listing
NOOBJ	Suppress object code generation
NOPAGE	Do not page listing file
PAGE	Begin new listing page
SPC	Generate blank lines in listing file
TTL	Set title in listing file

† Ignored by assembler.

Table 3.28: Output Directives

3.7.6. Asm68k Debugging Directives

Debugging directives are used to generate source-level debugging information. They are typically generated by the Sierra Systems C compiler to allow source-level debugging, but can also be used when programming in assembly language. Table 3.29 summarizes the debugging directives for **asm68k**.

The debugging directive descriptions are provided primarily to facilitate interpreting the compiler-generated directives. Their use is not recommended for programs written in assembly language. Instead, the **-L** command line flag should be used in conjunction with the **.type** directive to provide a reasonable level of debugging capability. When the **-L** command line flag is specified, a line number entry is generated for each instruction and memory allocation directive in text-type sections.

To accommodate the COFF object file format (see section **1.4.6 Line Number Information**), line number entries are associated with the most recently defined function (see **TYPE** directive); if no function has been defined, a dummy function with which to associate the entries is created. These dummy functions are named **^line1**, **^line2**, **^line3**, etc. The scope of both user-defined and dummy functions ends when either a new function is defined or a new section is created.

Directive	Function
.def / DEF	Begin symbol attribute block
.dim / DIM	Set array dimension attribute
.endef / ENDEF	End symbol attribute block
.file / FILE	Set name of source file
.line / LINE	Set line number attribute
.ln / LN	Create line number entry
.scl / SCL	Set storage class attribute
.size / SIZE	Set size attribute
.tag / TAG	Set tag name attribute
.type / TYPE	Set type attribute
.val / VAL	Set value attribute

Table 3.29: Debugging Directives

3.7.7. Asm68k Directive Reference

The remainder of this section provides, in alphabetical order, detailed descriptions of the directives supported by **asm68k**.

.align — Align Location Counter

Syntax

.align { 1 | 2 | 4 | 8 }

Description

The **.align** directive aligns the current section's location counter to the nearest multiple of the specified byte count. Any required padding is filled with the current fill value (see **.opt fillval**).

Example

```
.align 2
.word  0x4000
```

.bin / BIN — Include Contents of Binary File

Syntax

.bin ***filename***

BIN ***filename***

Description

filename The name of a binary file (including an optional absolute or relative path). It can optionally be enclosed in single or double quotes.

The **BIN** directive inserts the contents of the specified binary file at the current position in the assembler source. If the file is not specified with a full path, it is searched for in (or, relative to) the following directories in the indicated order:

1. The current directory.
2. Directories specified with the **`-I`** flag.
3. Directories specified with the environment variables **INCLUDE68** or **SIERRA** (see section **3.2.4 Environment Variables)**.

Examples

```
BIN "table.inc"
BIN "../include/graphics.seg"
```

.bsection / BSECTION — Begin / Resume a BSS-type Section

Syntax

.bsection *name* [, *address*]

BSECTION[.S] *name* [, *address*]

Description

name Specifies the section. It is a symbol whose name can be up to eight characters in length. The first character of the name can be a numerical character (0 – 9). If the section name begins with a numerical character, the assembler will prefix it with a period (.). Section names 9, 13, and 14, however, are handled differently; they designate the sections **.text**, **.data**, and **.bss**, respectively.

address Specifies the base or continuation address of the section. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.bsection / BSECTION** directive begins a BSS-type section with name *name*. If *address* is specified, the section is absolute and begins at that address; otherwise, the section is relocatable.

If the specified section already exists, it is resumed either at its current location (i.e., the value of its location counter) or at the specified address *address*. An absolute section can be restarted at any address beyond its current location. Relocatable sections cannot be resumed with an address specification.

The **.S** qualifier is used to indicate that any symbols defined in the section can be referenced with the absolute short addressing mode (i.e., the section will reside in the top or bottom 32K of memory).

No object code can be generated in BSS-type sections; they contain only uninitialized read/write data.

Examples

```
BSECTION  abc                ; relocatable
BSECTION  xyz,$4000          ; absolute
```

.bss — Begin / Resume the BSS-type Section .bss

Syntax

.bss

Description

The **.bss** directive begins or resumes the BSS-type section **.bss**. It is functionally equivalent to **.bsection .bss**.

The section **.bss** contains only uninitialized data; therefore, no object code can be generated in this section. The section **.bss** is present in all object files, regardless of whether it is specified. For more information, see section **3.3.4.1 Section Types**.

.byte — Generate Integer Data (Byte)

Syntax

.byte ***operand*** **[,** ***operand*****] . . .**

Description

operand Specifies an integer expression.

The **.byte** directive generates byte integer data. The values of the specified operands are placed in successive bytes beginning at the current location in the current section. The **.byte** directive accepts character strings delimited by double quotes. Escaped characters are also allowed in both character constants and strings. For additional information, see section **3.3.6.2 Character Constants**.

Examples

```
.byte "Hello!\0"                ; 4865 6C6C 6F21 00
.byte 'a'                       ; 61
.byte %1111,017,15,0xF          ; 0F0F 0F0F
.byte 16*4+3                    ; 43
```

Each of the above examples is shown with the code sequence (in hexadecimal) that it generates.

.cmnt — Begin Comment Block

Syntax

.cmnt

Description

The **.cmnt** directive begins a comment block. All assembler statements between this directive and its matching **.endc** directive are ignored. Pairs of comment block directives can be nested.

Example

```
.cmnt
    These lines are ignored by
    the assembler; no other
    comment markers are needed.
.endc
```

COMLINE — Allocate a Block of Uninitialized Memory

Syntax

COMLINE ***count***

Description

count Specifies the number of bytes to allocate. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **COMLINE** directive is functionally equivalent to the **DS.B** directive. It allocates a block of uninitialized memory whose size is determined by *count*. Each byte is filled with the current fill value (see **OPT FILLVAL**), unless the section is of BSS-type. This directive's intended use is not applicable to **asm68k**; it is supported only for Motorola compatibility.

.comm / COMM — Define a comm Symbol

Syntax

.comm *symbol*, *count* [, *align*]

COMM[.S] *symbol*, *count* [, *align*]

Description

symbol Specifies a symbol that is not defined elsewhere in the current file.

count Specifies the number of bytes associated with the symbol. It is an absolute expression that cannot contain any forward, external, or undefined references.

align Specifies the alignment requirements for the symbol. Its value can be 1, 2, or 4. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **COMM** directive defines the specified symbol *symbol* and associates with it a block of uninitialized data in the BSS-type section **.bss**. The number of bytes in this block is specified by *count*. The alignment of the block is specified by *align*; if omitted or if the **-c** flag has been specified, quad alignment is used.

The scope of the symbol *symbol* is external. The block of data is allocated during linkage, unless the **-6** flag has been specified, in which case it is allocated during assembly. For more information, see section **3.3.5.4 Comm and Lcomm Symbols**.

The **.S** qualifier is used to indicate that the symbol can be referenced with the absolute short addressing mode (i.e., the symbol will reside in the top or bottom 32K of memory).

Examples

```
COMM   table,4096
COMM   strings,256,2
COMM.S buffer,4
```

.data — Begin / Resume the Data-type Section .data

Syntax

.data

Description

The **.data** directive begins or resumes the data-type section **.data**. It is functionally equivalent to **.dsection .data**.

The section **.data** contains initialized read/write data. It is present in all object files, regardless of whether it is specified. For additional information, refer to section **3.3.4.1 Section Types**.

DC — Generate Integer / Floating-Point Data

Syntax

DC[*.size*] *operand* [, *operand*] . . .

Description

size Specifies the size and type of the data. The legal size qualifiers are shown below; the default is **W**.

- **B** Byte Integer (1 byte)
- **W** Word Integer (2 bytes)
- **L** Long-word Integer (4 bytes)
- **S** TI BCD floating point (10 bytes)
- **D** TI BCD floating point (10 bytes)
- **X** Extended-precision Real (not supported)
- **P** Packed decimal Real (not supported)

operand Specifies either an integer or floating-point value. Forward references are not allowed when specifying floating-point data.

The **DC** directive can be used to generate both integer and floating-point data. The values of the specified operands are placed in succession beginning at the current location in the current section. Word alignment of the data is forced except when *size* is **B**. For additional information, see section **3.3.6.2 Character Constants**.

Examples

```
DC.B 'abcd'               ; 6162 6364
DC.L $FF<<24              ; FF00 0000
DC.D 2.71828182846        ; 4000 2718 2818 2846 0000
```

Each of the above examples is shown with the sequence of words (in hexadecimal) that it generates.

DCB — Allocate a Block of Initialized Memory

Syntax

DCB[*.size*] *count*, *value*

Description

size Specifies the unit size. The legal size qualifiers are shown below; the default is **W**.

- **B** Byte Integer (1 byte)
- **W** Word Integer (2 bytes)
- **L** Long-word Integer (4 bytes)
- **S** Single-precision Real (not supported)
- **D** Double-precision Real (not supported)
- **X** Extended-precision Real (not supported)
- **P** Packed decimal Real (not supported)

count Specifies the unit count. It is an absolute expression that cannot contain any forward, external, or undefined references.

value Specifies the unit value. If *size* is **W** or **L**, any integer expression can be used; otherwise, an absolute expression that contains no forward, external, nor undefined references must be used.

The **DCB** directive allocates a block of initialized memory whose size is determined by the number and size of the unit location. Each location is filled with the specified value *value*. This directive cannot be used in BSS-type sections.

Examples

```
DCB   256,$FF
DCB.L 16,err_vector
```

.def / DEF — Begin Symbol Attribute Block

Syntax

.def *symbol*

DEF *symbol*

Description

symbol Specifies a symbol that is defined in the current assembler source file.

The **.def / DEF** directive begins an attribute block for the symbol *symbol*. The **DIM**, **LINE**, **SCL**, **SIZE**, **TAG**, **TYPE**, and **VAL** (**.dim**, **.line**, **.scl**, **.size**, **.tag**, **.type**, and **.val**) directives are used to set the various attributes of a symbol (see section **1.4 Object File Format**). The **.endef / ENDEF** directive must be used to end an attribute block. For convenience, the directives that comprise a symbol attribute block can be specified on a single line as a semicolon-separated list.

The information contained in an attribute block is stored in the object file's symbol table for purposes of symbolic debugging. The Sierra C compiler automatically generates attribute blocks for all symbols when the **-q** command line flag is specified. They can be written manually when performing assembler source-level debugging, but this is not recommended. Adequate debugging information can be generated with the **-L** assembler command line flag, which directs the assembler to generate line number information, and the **.type / TYPE** directive, which can be used to specify symbol types directly (i.e., without a symbol attribute block).

Examples

```
DEF init
VAL init
SCL 2
TYPE $24
ENDEF

DEF tbl ; VAL 12 ; SCL 3 ; TYPE 4 ; ENDEF
```

.dim / DIM — Set Array Dimension Attribute

Syntax

.dim ***dim*** **[,** ***dim*** **[,** ***dim*****[,** ***dim*** **]]]**

DIM ***dim*** **[,** ***dim*** **[,** ***dim*****[,** ***dim*** **]]]**

Description

dim — Specifies an array dimension of the current symbol. It is an absolute expression that cannot contain any forward, external, or undefined references. Up to four dimensions can be specified.

The **.dim / DIM** directive sets the dimension attribute of the symbol referenced by the current attribute block (see **.def / DEF** directive). The dimension attribute is specified for array types (see sections **1.4.8.7 Type Entry** and **1.4.9.6 Arrays**). This directive can appear at most once per symbol attribute block.

The **.dim / DIM** directive is typically used only for C source-level debugging; it is ignored when assembler source-level debugging information is generated with the **-L** command line flag (see also **.type / TYPE** directive).

Example

```
DEF buf
VAL buf
DIM 16,4              ; buf is a two-dim array, int buf[16][4]
SCL 2
TYPE $F4
LINE 25
SIZE 256
ENDEF
```

.double — Generate Floating-Point Data

Syntax

.double ***operand*** **[,** ***operand*****] . . .**

Description

operand Specifies a floating-point symbol or floating-point constant. No forward references are allowed.

The **.double** directive generates TI BCD floating-point data. The values of the specified operands are placed in 10 bytes beginning at the current location in the current section. A warning is issued if the alignment is odd.

Examples

```
_PI:
   .double 3.141592653589793       ; 4000 3141 5926 5358 9793
   .double 0x40003141592653589793  ; 4000 3141 5926 5358 9793
   .double -10,0.2                 ; C001 1000 0000 0000 0000
                                   ; 3FFF 2000 0000 0000 0000
```

Each of the above examples is shown with the sequence of words (in hexadecimal) that it generates.

DS — Allocate a Block of Uninitialized Memory

Syntax

DS[*.size*] *count*

Description

size Specifies the unit size. The legal size qualifiers are shown below; the default is **W**.

- **B** Byte Integer (1 byte)
- **W** Word Integer (2 bytes)
- **L** Long-word Integer (4 bytes)
- **S** Single-precision Real (not supported)
- **D** Double-precision Real (not supported)
- **X** Extended-precision Real (not supported)
- **P** Packed Decimal Real (not supported)

count Specifies the unit count. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **DS** directive allocates a block of uninitialized memory whose size is determined by the number and size of the unit location. Each location is filled with the current fill value (see **OPT FILLVAL**), unless the section is of BSS-type.

Specifying a count of zero forces alignment for the selected data size (e.g., **DS.W 0** forces word alignment).

Examples

```
DS   256
DS.L 64
```

.dsection / DSECTION — Begin / Resume a Data-type Section

Syntax

.dsection *name* [, *address*]

DSECTION[.S] *name* [, *address*]

Description

name Specifies the section. It is a symbol whose name can be up to eight characters in length. The first character of the name can be a numerical character (0 – 9). If the section name begins with a numerical character, the assembler will prefix it with a period (.). Section names 9, 13, and 14, however, are handled differently; they designate the sections **.text**, **.data**, and **.bss**, respectively.

address Specifies the base or continuation address of the section. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.dsection / DSECTION** directive begins a data-type section with name *name*. If *address* is specified, the section is absolute and begins at that address; otherwise, the section is relocatable.

If the specified section already exists, it is resumed either at its current location (i.e., the value of its location counter) or at the specified address *address*. An absolute section can be restarted at any address beyond its current location; any space that is created is filled with the current fill value (see **OPT FILLVAL**). Relocatable sections cannot be resumed with an address specification.

The **.S** qualifier is used to indicate that any symbols defined in the section can be referenced with the absolute short addressing mode (i.e., the section will reside in the top or bottom 32K of memory).

Examples

```
DSECTION table            ; relocatable
DSECTION ram,$8000        ; absolute
```

.echo — Echo Message

Syntax

.echo { *text* | *expression*[*.format*] } . . .

Description

text Specifies an ASCII string. Whitespace characters are permitted, but curly braces ({ }) are not.

expression Specifies an absolute expression that cannot contain any forward, external, or undefined references. The expression must be enclosed in curly braces ({ });

format Specifies the format for printing the preceding expression. The legal format characters, which correspond to those of the ANSI standard C library function **printf**, are shown below; the default is **d**.

- **d** Signed decimal
- **u** Unsigned decimal
- **o** Octal
- **x** Hexadecimal (lowercase)
- **X** Hexadecimal (uppercase)

The **.echo** directive causes the specified message to be written to the standard output stream, **stdout**. The end of the message is marked by the newline character; therefore, no comment field is permitted with this directive. This directive can be used for both informational and diagnostic purposes.

Example

```
.echo The end of the data section is {data_end}.x
```

The symbol data_end is a user-defined label, which has been defined prior to the directive in this example. If the value of the label is 0x42f8, then this directive produces the following message:

```
The end of the data section is 42f8
```

.elifdef — Assemble If Alternative Symbol Defined

Syntax

.elifdef *symbol*

Description

symbol Specifies a user-defined symbol

The **.elifdef** directive is used in a conditional assembly block. It is equivalent to the **.else** directive followed by an **.ifdef–.endif** block (see **.ifdef** and **.endif** directives). Its use obviates the need for multiple **.endif** directives in conditional blocks that have multiple alternatives.

The **.elifdef** directive is optional within a conditional assembly block.

Example

```
.ifdef option1
      moveq    #1,d0
.elifdef option2
      moveq    #2,d0
.elifdef option3
      moveq    #3,d0
.endif
```

.else — Assemble If Converse True

Syntax

.else

Description

The **.else** directive is used in a conditional assembly block. This directive matches the immediately preceding **.ifdef**, **.ifndef**, or **.elifdef** directive that is not matched by a **.endif** or **.else** directive. If this preceding directive fails, then the statements between the **.else** directive and the matching **.endif** directive are assembled; otherwise, these statements are skipped.

The **.else** directive is optional within a conditional assembly block.

Example

```
.ifdef debug
       moveq         #1,d0
.else
       moveq         #0,d0
.endif
```

ELSEC — Assemble If Converse True

Syntax

ELSEC

Description

The **ELSEC** directive is used in a conditional assembly block. This directive matches the immediately preceding **IFC**, **IFNC**, or **IFcc** directive that is not matched by an **ENDC** or **ELSEC** directive. If this preceding directive fails, then the statements between the **ELSEC** directive and the matching **ENDC** directive are assembled; otherwise, these statements are skipped.

The **ELSEC** directive is optional within a conditional assembly block.

Example

```
IFNE debug
       MOVEQ         #1,D0
ELSEC
       MOVEQ         #0,D0
ENDC
```

END — End Assembly

Syntax

END

Description

The **END** directive ends assembly in the current source file. Any assembler statements appearing after this directive are ignored.

.endc — End Comment Block

Syntax

.endc

Description

The **.endc** directive ends a comment block (see **.cmnt** directive). Note that the **.endc** directive is *not* the same as the **ENDC** directive.

ENDC — End Conditional Assembly Block

Syntax

ENDC

Description

The **ENDC** directive ends a conditional assembly block (see **IFC**, **IFNC**, and **IFcc** directives).

Example

```
IFNE debug
    JSR  mem_dump
ENDC
```

.endef / ENDEF — End Symbol Attribute Block

Syntax

.endef

ENDEF

Description

The **.endef / ENDEF** directive ends the current symbol attribute block (see **.def / DEF** directive).

.endif — End Conditional Assembly Block

Syntax

.endif

Description

The **.endif** directive ends a conditional assembly block (see **.ifdef** and **.ifndef** directives).

Example

```
.ifdef debug
    jsr  mem_dump
.endif
```

EQU — Define an Integer Symbol

Syntax

symbol* EQU *value

Description

symbol Specifies a symbol that is not defined elsewhere in the current file. It must appear in the label field of the statement.

value Specifies the value of the symbol. It is an absolute or simple relocatable expression that cannot contain any forward, external, or undefined references.

The **EQU** directive defines the symbol *symbol* and assigns to it the value *value*. The symbol is defined with static scope (the **XDEF** directive can be used to expand its scope to external). For more information, see **SET** directive and refer to section **3.3.5.3 Symbol Assignment**.

Examples

```
BUF_SIZE EQU 1024
TOT_LEN  EQU SYM_LEN+STR_LEN
```

.extend — Generate Floating-Point Data (Extended-Precision)

Not supported by Texas Instruments. However, **.extend** is still recognized as a reserved name by **asm68k**.

FAIL — Generate Error Message

Syntax

FAIL *code*

Description

code Specifies an error code. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **FAIL** directive causes the assembler to generate an error message. The value of *code* can be used to identify the offending directive or to provide useful diagnostic information. This directive is typically used in conditional assembly blocks — especially within macros — to mark a case that should not occur.

Examples

```
FAIL 83
FAIL \2                    ; second argument inside a macro
```

FEQU — Define a Floating-Point Symbol

Not supported by Texas Instruments. However, **FEQU** is still recognized as a reserved name by **asm68k**.

.file — Set Name of Source File

Syntax

.file "*filename*"

Description

filename Specifies the name of either the assembler source file or the corresponding C source file. It can be up to 14 characters in length.

The **.file** directive sets the name of the source file for purposes of source-level debugging. This directive can appear at most once per source file. It is generated internally by the assembler if it does not appear or if it specifies a C source file when the **-L** command line flag is used. For more information, see section **1.4.7.1 Special Symbols**.

Example

```
.file  "demo.c"
```

.float — Generate Floating-Point Data

Syntax

.float ***operand*** **[,** ***operand*****] . . .**

Description

operand Specifies a floating-point symbol or floating-point constant. No forward references are allowed.

The **.float** directive generates TI BCD floating-point data. The values of the specified operands are placed in 10 bytes beginning at the current location in the current section. A warning is issued if the alignment is odd. The data generated is the same as the **.double** directive, allowing 16 digits in the mantissa. Since a float in the compiler, **com68**, contains only 14 significant digits in the mantissa, it is recommended to always use **.double** to avoid confusion.

Examples

```
.float 3.141592653589793   ; 4000 3141 5926 5358 9793
.float -10,0.2             ; C001 1000 0000 0000 0000
                           ; 3FFF 2000 0000 0000 0000
```

Each of the above examples is shown with the sequence of words (in hexadecimal) that it generates.

FOPT — Set Assembler Floating-Point Options

Not supported by Texas Instruments. However, **FOPT** is still recognized as a reserved name by **asm68k**.

FORMAT — Format Assembly Listing

Syntax

FORMAT

Description

The **FORMAT** directive is ignored. It is recognized only for Motorola compatibility.

.global / .globl — Declare External a Defined Symbol

Syntax

.global *symbol* [, *symbol*] . . .

.globl *symbol* [, *symbol*] . . .

Description

symbol Specifies a symbol that is defined in the current file. Section names and floating-point symbols are not allowed.

The **.global** and **.globl** directives declare the scope of the symbol *symbol* to be external. This is necessary when the symbol is referenced in other source files since the default symbol scope is static. This directive is a synonym for the **XDEF** directive.

Example

```
.global   main,jmp_tbl
.globl    procl,eval
```

IDNT — Set Name of Source File

Syntax

***filename* IDNT**

Description

filename Specifies the name of the assembler source file. It can be up to 14 characters in length and must appear in the label field of the statement.

The **IDNT** directive sets the name of the source file for purposes of source-level debugging. This directive can appear at most once per source file and is generated internally by the assembler if omitted. For more information, refer to the discussion of the **.file** symbol in section **1.4.7.1 Special Symbols**.

Example

```
demo.s IDNT
```

IFC — Assemble If Strings Equal

Syntax

IFC '*string1*', '*string2*'

Description

string1 Specifies an ASCII string. It can include whitespace and commas.

string2 Specifies an ASCII string. It can include whitespace and commas.

The **IFC** directive introduces a conditional assembly block. If the specified strings *string1* and *string2* are equal, then the statements between this directive and the first matching **ELSEC** or **ENDC** directive are assembled, and the remainder of the block is skipped. Otherwise, the statements associated with the **IFC** directive are skipped and control passes to the aforementioned matching directive.

The **IFC** directive is useful only within user-defined macros (e.g., testing for null parameters). The single quotes that enclose each string need not appear literally in the operand field of this directive prior to macro expansion — i.e., they can be included in an actual parameter.

Conditional assembly directives can be nested to 40 levels.

Example

```
IFC '\2',''
    MOVE.L  #1,match
ENDC
```

IFcc — Assemble If Condition True

Syntax

IF*cc value*

Description

cc Specifies a conditional relation between the specified value and zero. The following are the valid condition codes:

- **EQ** Expression is equal to zero
- **NE** Expression is not equal to zero
- **GE** Expression is greater than or equal to zero
- **GT** Expression is greater than zero
- **LE** Expression is less than or equal to zero
- **LT** Expression is less than zero

value Specifies the control value for the assembly block. It is an absolute expression that cannot contain any forward, external, or undefined references.

Each **IFcc** directive introduces a conditional assembly block. If the value *value* observes the specified conditional relation, then the statements between the given **IFcc** directive and the first matching **ELSEC** or **ENDC** directive are assembled and the remainder of the block is skipped. Otherwise, the statements associated with the **IFcc** directive are skipped and control passes to the aforementioned matching directive.

Conditional assembly directives can be nested to 40 levels.

Example

```
IFGT count
    MOVEQ  #5,D2
ENDC
```

.ifdef — Assemble If Symbol Defined

Syntax

.ifdef *symbol*

Description

symbol Specifies a user-defined symbol.

The .**ifdef** directive introduces a conditional assembly block. If the specified symbol *symbol* is defined when the directive is encountered, then the statements between this directive and the first matching **.elifdef**, **.else**, or **.endif** directive are assembled and the remainder of the block is skipped. Otherwise, the statements associated with the **.ifdef** directive are skipped and control passes to the aforementioned matching directive.

Conditional assembly directives can be nested to 40 levels.

Example

```
.ifdef serial
    move.l  #serial_dev,io_func
.endif
```

IFNC — Assemble If Strings Not Equal

Syntax

IFNC '*string1*', '*string2*'

Description

string1 Specifies an ASCII string. It can include whitespace and commas.

string2 Specifies an ASCII string. It can include whitespace and commas.

The **IFNC** directive introduces a conditional assembly block. If the specified strings *string1* and *string2* are not equal, then the statements between this directive and the first matching **ELSEC** or **ENDC** directive are assembled, and the remainder of the block is skipped. Otherwise, the statements associated with the **IFNC** directive are skipped and control passes to the aforementioned matching directive.

The **IFNC** directive is useful only within user-defined macros (e.g., testing for null parameters). The single quotes that enclose each string need not appear literally in the operand field of this directive prior to macro expansion — i.e., they can be included in an actual parameter.

Conditional assembly directives can be nested to 40 levels.

Example

```
IFNC '\1',''
    MOVE.W  D0,D3
ENDC
```

.ifndef — Assemble If Symbol Not Defined

Syntax

.ifndef *symbol*

Description

symbol Specifies a user-defined symbol.

The **.ifndef** directive introduces a conditional assembly block. If the specified symbol *symbol* is not defined when the directive is encountered, then the statements between this directive and the first matching **.elifdef**, **.else**, or **.endif** directive are assembled and the remainder of the block is skipped. Otherwise, the statements associated with the **.ifdef** directive are skipped and control passes to the aforementioned matching directive.

Conditional assembly directives can be nested to 40 levels.

Example

```
.ifndef serial
    move.l  #parallel_dev,io_func
.endif
```

.include / INCLUDE — Include Assembler Source File

Syntax

.include *filename*

INCLUDE *filename*

Description

filename The name of an assembler source file (including an optional absolute or relative path). It can optionally be enclosed in single or double quotes.

The **INCLUDE** directive inserts the contents of the specified file at the current position in the assembler source. If the file is not specified with a full path, it is searched for in (or, relative to) the following directories in the indicated order:

1. The current directory.
2. Directories specified with the **`-I`** flag.
3. Directories specified with the environment variables **INCLUDE68** or **SIERRA** (see section **3.2.4 Environment Variables**).

This directive can be nested, i.e., included files can themselves include files. The assembler imposes no limit on the level of nesting.

Examples

```
INCLUDE "table.inc"
INCLUDE "../include/vector.h"
```

.lcomm / LCOMM — Define an lcomm Symbol

Syntax

.lcomm *symbol*, *count* [, *align*]

LCOMM[.S] *symbol*, *count* [, *align*]

Description

symbol Specifies a symbol that is not defined elsewhere in the current file.

count Specifies the number of bytes associated with the symbol. It is an absolute expression that cannot contain any forward, external, or undefined references.

align Specifies the alignment requirements for the symbol. Its value can be 1, 2, or 4. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **LCOMM** directive defines the specified symbol *symbol* and associates with it a block of uninitialized data in the BSS-type section **.bss**. The number of bytes in this block is specified by *count*. The alignment of the block is specified by *align*; if omitted or if the **-c** flag has been specified, quad alignment is used.

The scope of the symbol *symbol* is static. The block of data is allocated during assembly. For more information, refer to section **3.3.5.4 Comm and Lcomm Symbols**).

The **.S** qualifier is used to indicate that the symbol can be referenced with the absolute short addressing mode (i.e., the symbol will reside in the top or bottom 32K of memory).

Examples

```
LCOMM table,4096
LCOMM strings,256,4
```

.line / LINE — Set Line Number Attribute

Syntax

.line ***line***

LINE ***line***

Description

line Specifies the number of the line on which the current symbol is defined. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.line / LINE** directive sets the line number attribute of the symbol referenced by the current attribute block (see **.def / DEF** directive). For a detailed description of this attribute, see section **1.4.9 Auxiliary Table Entries**. This directive can appear at most once per symbol attribute block.

The **.line / LINE** directive is typically used only for C source-level debugging; it is ignored when assembler source-level debugging information is generated with the **-L** command line flag (see also **.type / TYPE** directive).

Example

```
DEF buf
VAL buf
DIM 16,4
SCL 2
TYPE $F4
LINE 25                    ; buf is defined on line 25
SIZE 256
ENDEF
```

LIST — Enable Assembly Listing

Syntax

LIST

Description

The **LIST** directive causes a listing of the assembly to be generated (default). It is used to resume the assembly listing after it has been disabled with the **NOL** or **NOLIST** directive. Pairs of **NOLIST** and **LIST** directives can be nested.

LLEN — Set Line Length of Listing File

Syntax

LLEN ***length***

Description

length	Specifies the line length. It is an absolute expression that cannot contain any forward, external, or undefined references. The minimum line length is 40.

The **LLEN** directive sets the line length of the listing file to *length* characters. The default line length is 80.

Example

```
LLEN 132
```

.ln / LN — Create Line Number Entry

Syntax

.ln ***line***

LN ***line***

Description

line Specifies the current line number in the source file. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.ln / LN** directive creates a line number entry for purposes of source-level debugging. This entry associates the current section's location counter with a line in the associated C source file. For more information, see section **1.4.6 Line Number Information**. This directive cannot be used in BSS-type sections.

The **.ln / LN** directive is typically used only for C source-level debugging; it is ignored when assembler source-level line number information is generated with the **-L** command line flag.

Example

```
LN 33
ADD D0,D1
```

.long — Generate Integer Data (Long-Word)

Syntax

.long ***operand*** **[,** ***operand*****] . . .**

Description

operand Specifies an integer expression.

The **.long** directive generates long-word integer data. The values of the specified operands are placed in successive long words beginning at the current location in the current section. A warning is issued if the alignment is odd. For additional information, see section **3.3.6.2 Character Constants**.

Examples

```
.long 'abcd'                    ; 61626364
.long 16777215,0xFFFFFF         ; 00FFFFFF 00FFFFFF
.long 0xFF<<24                  ; FF000000
```

Each of the above examples is shown with the sequence of long words (in hexadecimal) that it generates.

MASK2 — Assemble For Mask2 Chip

Syntax

MASK2

Description

The **MASK2** directive is ignored. It is recognized only for Motorola compatibility.

NOFORMAT — Do Not Format Assembly Listing

Syntax

NOFORMAT

Description

The **NOFORMAT** directive is ignored. It is recognized only for Motorola compatibility.

NOL / NOLIST — Disable Assembly Listing

Syntax

NOL

NOLIST

Description

The **NOL** and **NOLIST** directives prevent a listing of the assembly from being generated. This allows portions of the assembly to be omitted from the listing file. The listing is resumed with the **LIST** directive. Pairs of **NOLIST** and **LIST** directives can be nested.

NOOBJ — Suppress Object Code Generation

Syntax

NOOBJ

Description

The **NOOBJ** directive suppresses generation of an object file. This directive has the same effect as the **-K** command line flag.

NOPAGE — Do Not Page Listing File

Syntax

NOPAGE

Description

The **NOPAGE** directive suppresses paging in the listing file. The listing is generated as one continuous page.

OFFSET — Begin a Structure Template Section

Syntax

OFFSET [*address*]

Description

address Specifies the base address of the section. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **OFFSET** directive begins a structure template section. If *address* is specified, the section begins at that address; otherwise, it begins at absolute address zero. This type of section is used in conjunction with the **DS** directive to define labels suitable for structure field references. These labels are not included in the object file's symbol table.

Since structure template sections are dummy sections, they cannot contain any object code. Also, they cannot be nested; however, fields of a nested structure can be treated as part of the enclosing structure. Any section directive will end a structure template section.

Example

```
            OFFSET                      ; struct node {
visited     DS.B        1               ;    char visited;
            DS.W        0               ;    struct position {
pos.x       DS.B        2               ;       short int x;
pos.y       DS.B        2               ;       short int y;
            DS.W        0               ;    } pos;
left        DS.B        4               ;    struct node *left;
right       DS.B        4               ;    struct node *right;
            ORG         $1000           ; };
            MOVE.B      #1,_n+visited   ; n.visited = 1;
            MOVEA.L     _p,A0           ; p->right = NULL;
            CLR.L       (right,A0)      ;
```

This example illustrates how the **OFFSET** directive is used to define a set of structure field labels. The **DS** directive is used to allocate space and maintain proper alignment.

.opt / OPT — Set Assembler Options

Syntax

.opt ***option*** **[,** ***option*****] . . .**

OPT ***option*** **[,** ***option*****] . . .**

Description

option An assembler option (see complete list below).

The **.opt / OPT** directive sets the specified assembler options. These options affect the assembly of instructions, effective addresses, and data. They also provide a means of customizing the assembly listing output. Only options valid for the 68000 are supported by Texas Instruments.

Options

BRB / BRS
BRW
Set the size of unknown PC-relative displacements to 8 or 16 bits (see section **3.5.4.2 Displacement Sizing**).
(Default: **BRW**)

CASE
NOCASE
Enable/disable character case-sensitivity in symbol names (see section **3.3.5.1 Symbol Syntax**).
(Default: **CASE**)

CEX
NOCEX
Enable/disable full listing of data directive assembly. The **CEX** option allows the object code associated with the **DC** directive to be listed in its entirety, while the **NOCEX** option permits only one line of object code to be listed. Pairs of these complement options can be nested.
(Default: **CEX**)

CHOP
NOCHOP
Enable/disable truncation of lines in the assembly listing (see **-w** and **-x** command line flags).
(Default: **NOCHOP**)

CL
NOCL
Enable/disable listing of conditional assembly directives (see **IFC**, **IFNC**, and **IFcc** directives).
(Default: **CL**)

CRE
This option is ignored. It is recognized only for Motorola compatibility.

D
This option is ignored. It is recognized only for Motorola compatibility.

EQU
NOEQU
Enable/disable inclusion of equate symbols (i.e., symbols created via the **EQU** and **SET** directives) in the object file's symbol table.
(Default: **EQU**)

FILLVAL=*value*
: Set the fill value to *value* (see **DS** directive). A warning is issued if the specified value does not fit in a signed or unsigned byte.
(Default: **FILLVAL=0**)

Flash ROM 16
FR32
: Set the size of unknown PC-relative displacements to 16 or 32 bits (not supported by Texas Instruments).

FRS
FRL
: Set the size of unknown absolute displacements to either 16 or 32 bits (see section **3.5.4.2 Displacement Sizing**).
(Default: **FRL**)

FWDSIZE
: Enable displacement size checking for forward branches. The assembler will issue a warning for each displacement that can be reduced in size.
(Default: **disabled**)

IOPT
NOIOPT
: Enable/disable instruction optimizations (see section **3.4.3 Instruction Optimization**).
(Default: **IOPT**)

ISIZE=*size*
: Set the default instruction size (see section **3.4.2 Instruction Sizing**). The following are the legal sizes:

  B Byte Integer

  W Word Integer

  L Long-word Integer

  (Default: **ISIZE=W**)

LHEX
: List the alphabetical hexadecimal digits using lowercase characters.
(Default: **UHEX**)

LLBL
NOLLBL
: Enable/disable local labels (see section **3.3.5.2 Labels**).
(Default: **NOLLBL**)

MC
NOMC
: Enable/disable listing of user-defined macro invocations (see section **3.8.1.2 Macro Invocation**).
(Default: **MC**)

MD
NOMD
: Enable/disable listing of user-defined macro definitions (see section **3.8.1.1 Macro Definition**).
(Default: **MD**)

MEX
NOMEX
: Enable/disable listing of user-defined macro expansions (see section **3.8.1 User-Defined Macros**).
(Default: **NOMEX**)

NOPC
: Disable all coercions to PC-relative addressing modes. This option disables the **PCB16**, **PCB32**, and **PCF** options.
(Default: **PCB16**)

O
NOO
: Enable/disable generation of the object file. This option is overridden by both the **NOOBJ** directive and the **-K** flag.
(Default: **O**)

OLD
NOOLD
Enable/disable old branch sizing (see section **3.5.4.2 Displacement Sizing**).
(Default: **NOOLD**)

P=*proc*
PROC=*proc*
Set the target processor. **68000** is the only processor recognized by Texas Instruments.
(Default: **P=68000**)

PCA
NOPCA
Enable/disable coercion to PC-relative addressing modes for references to an absolute location or from an absolute section. The **PCF** option must be enabled for this option to be effective (see section **3.5.4.1 PC-relative Coercion**).
(Default: **PCA**)

PCB16
PCB32
Enable coercion to PC-relative addressing modes for 16-bit and 32-bit backward references (see section **3.5.4.1 PC-relative Coercion**). Selecting the **PCB16** option disables the **PCB32** option; selecting the **PCB32** option enables the **PCB16** option. The **PCB32** option is not legal on the 68000/10.
(Default: **PCB16**)

PCF
Enable coercion to PC-relative addressing modes for forward references, references to locations positioned at an unknown distances. This option also enables the **PCB16** option. For more information, see section **3.5.4.1 PC-relative Coercion**.
(Default: **PCB16**)

RNGCHK [=] *type*
Set the immediate data range check for byte and word data. The following four levels of range checking are provided:

0 No checking: all values are silently truncated

1 Byte: -256–255; Word: -65536–65535

2 Byte: -128–255; Word: -32768–65535

3 Byte: -128–127; Word: -32768–32767

(Default: **RNGCHK=1**)

SCEX
NOSCEX
Enable/disable listing of structured control macro expansions (see section **3.8.2 Structured Control Macros**).
(Default: **NOSCEX**)

UHEX
List the alphabetical hexadecimal digits using uppercase characters (see also **LHEX**).
(Default: **UHEX**)

ULOC
Modify the naming convention for compiler locals by adding an initial underscore (_). For more information, see section **3.3.5.6 Compiler Locals**.
(Default: **disabled**)

ORG — Begin an Absolute Data-type Section

Syntax

ORG[*.size*] *address*

Description

size Specifies the size of unknown absolute displacements contained in the section. The legal values are shown below; the default is **L**.

- **S** Absolute short reference (16-bit)
- **L** Absolute long reference (32-bit)

address Specifies the base address of the section. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **ORG** directive begins an unnamed, data-type section. The section's base address *address* is fixed.

Examples

```
ORG.S $4000
ORG   base+1024
```

In the second example, the symbol **base** must be absolute and previously defined.

PAGE — Begin New Listing Page

Syntax

PAGE

Description

The **PAGE** directive begins a new page in the listing file. This directive does not appear in the listing.

REG — Define a Register List Symbol

Syntax

***symbol* REG *register* [–*register*] [/*register* [–*register*]] . . .**

Description

symbol Specifies a symbol that is not defined elsewhere in the current file. It must appear in the label field of the statement.

register Specifies a data, address, or floating-point register. Hyphen-separated ranges must be specified in ascending order.

The **REG** directive defines the symbol *symbol* and assigns to it the specified register list. Register list symbols are suitable for use with the **MOVEM** and **FMOVEM** instructions. For more information on register list specification, see section **3.5.3 Effective Address Syntax**.

Examples

```
SAVE    REG D3-D7/A2
```

REORG — Reset the Location Counter in an Absolute Section

Syntax

REORG *address*

Description

address Specifies the address at which the current section will continue. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **REORG** directive resets the location counter in an absolute section. The continuation address *address* must be greater than the current value of the section's location counter; any space that is created is filled with the current fill value (see **OPT FILLVAL**), unless the section is of BSS-type.

Examples

```
ORG.L    $8000
MOVE.L   #$5000,D0
REORG    $C000

BSECTION tbl,$1000
MOVE.L   #$1200,D1
REORG    $8000
```

As the second example illustrates, the **REORG** directive can be applied to any absolute section, regardless of its declaration.

.scl / SCL — Set Storage Class Attribute

Syntax

.scl ***class***

SCL ***class***

Description

class Specifies the storage class of the current symbol. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.scl / SCL** directive sets the storage class attribute of the symbol referenced by the current attribute block (see **.def / DEF** directive). For a list of recognized storage classes, see section **1.4.8.2 Storage Class**. This directive can appear at most once per symbol attribute block.

The **.scl / SCL** directive is typically used only for C source-level debugging; it is ignored when assembler source-level debugging information is generated with the **-L** command line flag (see also **.type / TYPE** directive).

Example

```
DEF x
VAL 3
SCL 4                    ; x is in a register
TYPE 4
ENDEF
```

SECTION — Begin / Resume a Data-type Section

Syntax

SECTION[.S] ***name*** **[,** ***address*****]**

Description

name Specifies the section. It is a symbol whose name can be up to eight characters in length. The first character of the name can be a numerical character (0–9). If the section name begins with a numerical character, the assembler will prefix it with a period (.). Section names 9, 13, and 14, however, are handled differently; they designate the sections **.text**, **.data**, and **.bss**, respectively.

address Specifies the base or continuation address of the section. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **SECTION** directive begins a data-type section with name *name*. If *address* is specified, the section is absolute and begins at that address; otherwise, the section is relocatable.

If the specified section already exists, it is resumed either at its current location (i.e., the value of its location counter) or at the specified address *address*. An absolute section can be restarted at any address beyond its current location; any space that is created is filled with the current fill value (see **OPT FILLVAL**). Relocatable sections cannot be resumed with an address specification.

The **.S** qualifier is used to indicate that any symbols defined in the section can be referenced with the absolute short addressing mode (i.e., the section will reside in the top or bottom 32K of memory).

This directive is a synonym for the **DSECTION** directive.

SET — Define / Redefine an Integer Symbol

Syntax

symbol* SET *value

Description

symbol Specifies a symbol that can be defined elsewhere in the current file only with the **SET** directive. It must appear in the label field of the statement.

value Specifies the value of the symbol. It is an absolute or simple relocatable expression that cannot contain any forward, external, or undefined references.

The **SET** directive defines the symbol *symbol* and assigns to it the value *value*. The symbol is defined with static scope (the **XDEF** directive can be used to expand its scope to external). If the symbol was previously defined with the **SET** directive, it is redefined with the specified value; its scope is not modified. For more information, see section **3.3.5.3 Symbol Assignment**.

Examples

```
COUNT  SET 1
COUNT  SET COUNT+1
COUNT  SET COUNT+1
```

.short — Generate Integer Data (Word)

Syntax

.short *operand* [, *operand*] . . .

Description

operand Specifies an integer expression.

The **.short** directive generates word integer data. The values of the specified operands are placed in successive words beginning at the current location in the current section. A warning is issued if the alignment is odd. For additional information, refer to section **3.3.6.2 Character Constants**.

Examples

```
.short 'ab'                    ; 6162
.short 01777,1023,0x3FF        ; 03FF 03FF 03FF
.short 1024*16                 ; 4000
```

Each of the above examples is shown with the sequence of words (in hexadecimal) that it generates.

.size / SIZE — Set Size Attribute

Syntax

.size ***size***

SIZE ***size***

Description

size Specifies the size of the current symbol. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.size / SIZE** directive sets the size attribute of the symbol referenced by the current attribute block (see **.def / DEF** directive). The size attribute is specified for aggregate types (see section **1.4.9 Auxiliary Table Entries**). This directive can appear at most once per symbol attribute block.

The **.size / SIZE** directive is typically used only for C source-level debugging; it is ignored when assembler source-level debugging information is generated with the **-L** command line flag (see also **.type / TYPE** directive).

Example

```
DEF buf
VAL buf
DIM 16,4
SCL 2
TYPE $F4
LINE 25
SIZE 256                    ; buf is an array 256 bytes long
ENDEF
```

.space — Allocate a Block of Uninitialized Memory

Syntax

.space[.*size*] *count*

Description

size Specifies the unit size. The legal sizes are shown below; the default is **b**.

- **b** Byte Integer (1 byte)
- **w** Word Integer (2 bytes)
- **l** Long-word Integer (4 bytes)
- **s** Single-precision Real (not supported)
- **d** Double-precision Real (not supported)
- **x** Extended-precision Real (not supported)
- **p** Packed Decimal Real (not supported)

count Specifies the unit count. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.space** directive allocates a block of uninitialized memory whose size is determined by the number and size of the unit location. Each location is filled with the current fill value (see **OPT FILLVAL**), unless the section is of BSS-type.

Examples

```
.space    256
.space.l  64
```

SPC — Generate Blank Lines in Listing File

Syntax

SPC *count*

Description

count Specifies the number of blank lines. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **SPC** directive generates *count* blank lines in the listing file. This directive does not appear in the listing.

Example

```
SPC  3
```

.tag / TAG — Set Tag Name Attribute

Syntax

.tag ***symbol***

TAG ***symbol***

Description

symbol Specifies the tag name with which the current symbol is associated. It is a symbol that is defined as a structure, union, or enumeration type.

The **.tag / TAG** directive sets the tag name attribute of the symbol referenced by the current attribute block (see **.def / DEF** directive). For a detailed description of the tag name attribute, see section **1.4.9 Auxiliary Table Entries**. This directive can appear at most once per symbol attribute block.

The **.tag / TAG** directive is typically used only for C source-level debugging; it is ignored when assembler source-level debugging information is generated with the **-L** command line flag (see also **.type / TYPE** directive).

Example

```
DEF x
VAL x
SCL 2
TYPE 8
SIZE 4
TAG s               ; s is the tag of the structure
ENDEF               ; of which x is a member
```

.text — Begin / Resume the Text-type Section .text

Syntax

.text

Description

The **.text** directive begins or resumes the text-type section **.text**. It is functionally equivalent to **.tsection .text**.

The section **.text** contains read-only data. It is present in all object files, regardless of whether it is specified. For more information, see section **3.3.4.1 Section Types**.

.tsection / TSECTION — Begin / Resume a Text-type Section

Syntax

.tsection *name* [, *address*]

TSECTION[.S] *name* [, *address*]

Description

name Specifies the section. It is a symbol whose name can be up to eight characters in length. The first character of the name can be a numerical character (0–9). If the section name begins with a numerical character, the assembler will prefix it with a period (.). Section names 9, 13, and 14, however, are handled differently; they designate the sections **.text**, **.data**, and **.bss**, respectively.

address Specifies the base or continuation address of the section. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.tsection / TSECTION** directive begins a text-type section with name *name*. If *address* is specified, the section is absolute and begins at that address; otherwise, the section is relocatable.

If the specified section already exists, it is resumed either at its current location (i.e., the value of its location counter) or at the specified address *address*. An absolute section can be restarted at any address beyond its current location; any space that is created is filled with the current fill value (see **OPT FILLVAL**). Relocatable sections cannot be resumed with an address specification.

The **.S** qualifier is used to indicate that any symbols defined in the section can be referenced with the absolute short addressing mode (i.e., the section will reside in the top or bottom 32K of memory).

Examples

```
TSECTION code             ; relocatable
TSECTION rom,$F800        ; absolute
```

TTL — Set Title in Listing File

Syntax

TTL ***title***

Description

title Specifies the title that appears on each listing file page. It is an ASCII string that can be optionally enclosed in double quotes.

The **TTL** directive sets the title that appears in the header of each page of the listing file. This directive stays in effect until another **TTL** directive is encountered; the current page is affected only if the directive appears on the page's first line. If this directive is not specified, the assembler's name is used as the title.

Example

```
TTL "macro assembler"
TTL Sierra Systems is #1
```

.type / TYPE — Set Type Attribute

Syntax

.type ***type***

TYPE ***type***

Description

type Specifies the type of the current symbol. It is an absolute expression that cannot contain any forward, external, or undefined references.

The **.type / TYPE** directive sets the type attribute of the symbol referenced by the current attribute block (see **.def / DEF** directive). For a discussion of fundamental and derived types, see section **1.4.8.7 Type Entry**. This directive can appear at most once per symbol attribute block.

The **.type / TYPE** directive can also be used in conjunction with the **-L** command line flag to provide type information for symbols. When the **-L** command line flag is used, this directive does not have to appear within a symbol attribute block; the type it specifies is associated with the most recently defined label. Function types can only be attributed to labels defined in text-type sections.

Examples

```
x:
TYPE $24            ; x is a function that returns an int
```

.val / VAL — Set Value Attribute

Syntax

.val ***value***

VAL ***value***

Description

value Specifies the value of the current symbol. It is an absolute or simple relocatable expression.

The **.val / VAL** directive sets the value attribute of the symbol referenced by the current attribute block (see **.def / DEF** directive). The meaning of the value attribute is dependent upon the selected storage class (see section **1.4.8.4 Symbol Value Field**). This directive can appear at most once per symbol attribute block.

The **.val / VAL** directive is typically used only for C source-level debugging; it is ignored when assembler source-level debugging information is generated with the **-L** command line flag (see also **.type / TYPE** directive).

Example

```
DEF x
VAL x              ; x is a variable — its value is its location
SCL 2
TYPE 4
ENDEF
```

.word — Generate Integer Data (Word)

Syntax

.word *operand* [, *operand*] . . .

Description

operand Specifies an integer expression.

The **.word** directive generates word integer data. The values of the specified operands are placed in successive words beginning at the current location in the current section. A warning is issued if the alignment is odd. For additional information, refer to section **3.3.6.2 Character Constants**.

Examples

```
.word 'ab'                          ; 6162
.word %1111111111,01777,1023        ; 03FF 03FF 03FF
.word 1024*16                       ; 4000
```

Each of the above examples is shown with the sequence of words (in hexadecimal) that it generates.

XDEF — Declare External a Defined Symbol

Syntax

XDEF *symbol* [, *symbol*] . . .

Description

symbol Specifies a symbol that is defined in the current file. Section names and floating-point symbols are not allowed.

The **XDEF** directive declares the scope of the symbol *symbol* to be external. This is necessary when the symbol is referenced in other source files (since the default symbol scope is static).

Examples

```
XDEF init
XDEF proc1,diag_list
```

XREF — Declare External a Referenced Symbol

Syntax

XREF[.S] [*section*:]*symbol* [, [*section*:]*symbol*] . . .

Description

section Specifies the number of the section in which the symbol is defined. It is an absolute expression that cannot contain any forward, external, or undefined references.

symbol Specifies a symbol that is referenced (but not defined) in the current file. Section names and floating-point symbols are not allowed.

The **XREF** directive declares the scope of the symbol *symbol* to be external. This is necessary when a locally referenced symbol is defined in another source file. Failure to declare such a symbol to have external scope results in an error (see also **-E** flag).

The **.S** qualifier is used to indicate that the symbol(s) can be referenced with the absolute short addressing mode. The optional section number is ignored; it is recognized only for Motorola compatibility.

Examples

```
XREF   base,init,input
XREF.S proc1
```

3.8. Asm68k Macros

The assembler **asm68k** supports two macro facilities: user-defined macros and structured control macros. They are provided to simplify the task of programming, improve the readability of assembly source, and help reduce the frequency of errors. This section describes the user-defined macro and structured control macro support.

3.8.1. User-Defined Macros

Macros provide an efficient means of generating commonly used sequences of code. A macro is defined once and can be used any number of times. Parameters can be used to allow variations in the code sequences generated by a single macro.

Using macros simplifies programming, since changes can be isolated to a single location. Also, assembly programs written using macros are more concise and easier to understand. The remainder of this section describes how to define and invoke macros, as well as how to use macro parameters and conditional control.

3.8.1.1. Macro Definition

A macro definition consists of a header, a body, and a terminator. Following is the syntax for a macro definition:

Syntax

label **MACRO [** ***comment*** **]**
[***statement*** **]**
.
.
.
ENDM

Description

label Specifies the name of the macro. The first 32 characters of the macro name are significant. Only the first character of the name can be a period (.).

comment Specifies a comment, which is typically used to document any formal parameters of the macro.

statement Specifies an assembler statement. It cannot be a macro definition (i.e., macro definitions cannot be nested).

The header consists of the macro name *label*, the **MACRO** directive, and an optional comment field. The terminator is the **ENDM** directive. The body is a sequence of statements that are assembled each time the macro is invoked. These statements can reference the formal parameters of the macro (see section **3.8.1.3 Parameters**).

A macro can be referenced from within another macro definition prior to being defined itself. Macros cannot be redefined, nor can they use the names of instructions or directives.

3.8.1.2. Macro Invocation

Following is the syntax for macro invocations:

Syntax

[*label*] *macro*[*.qualifier*] [*parameter* [, *parameter*] . . .]

Description

label	Specifies an optional label.
macro	Specifies the name of the macro. It must have been previously defined with the **MACRO** directive.
qualifier	Specifies a size qualifier that is passed to the macro as parameter **\0**.
parameter	Specifies symbols, constants, expressions, or any other text. The maximum number of parameters allowed is 35. They are passed to the macro as parameters **\1**, **\2**, **\3**, etc.

When a macro is invoked, it generates assembler statements according to its definition and actual parameters. These statements are treated as are any other source statements. Any nested macros will be expanded when they are encountered.

Once a macro is recognized, the specified actual parameters are saved. The first specified parameter corresponds to formal parameter **\1**, the second corresponds to **\2**, etc. Parameters are passed by name, not by value. Since the parameter-passing mechanism is simply textual substitution, values of symbols can be modified by the macro.

The following steps are then performed for each statement in the body of the macro:

- The statement is retrieved.
- Any formal parameters are replaced with their corresponding actual parameters; formal parameters that have no corresponding actual parameter are assigned a null string.
- The **\@** designator is replaced with a unique number (refer to section **3.8.1.4 Local Labels**).
- The **NARG** symbol is replaced with its value (see section **3.8.1.5 NARG Symbol**).
- The fully expanded statement is processed as any other statement would be processed. If it is a nested macro invocation, these steps are performed recursively.

Macros must be defined before they are invoked. A macro's expanded code will appear in the listing file only if the **OPT MEX** directive is specified (default: **OPT NOMEX**).

3.8.1.3. Parameters

The formal parameters of a macro are the parameters that are referenced in the macro's definition. They are denoted by a backslash (\ ) followed by a digit or alphabetical character (either lowercase or uppercase). The formal parameter designated by **\0** corresponds to the optional size qualifier used when the macro is invoked. The parameters designated by **\1 – \9** and **\A – \Z** correspond to the actual parameters that appear in the operand field of a macro invocation. If a macro is defined with a non-consecutive set of formal parameters (e.g., **\3** is referenced, but **\2** is not), a warning is issued.

The actual parameters of a macro are the parameters that are specified when a macro is invoked. The actual parameters are substituted for their formal counterparts in the body of the expanded macro. The default value for any parameter in the operand field is a null string, and the default for the parameter in the qualifier field is the size qualifier **.W**. If an actual parameter is omitted, its value is the default.

If an actual parameter contains whitespace or commas, it must be enclosed in angle brackets (< >). Angle brackets can be nested, but must always appear in pairs.

An actual parameter that begins with a question mark (?) is evaluated prior to macro expansion; the result of the integer expression becomes the actual parameter. The expression must be absolute and cannot contain any forward, external, or undefined references (see section **3.3.7.3 Expression Evaluation**). The question mark operator is necessary when a formal parameter is situated in a field where an expression cannot be specified (e.g., within in a symbol name). It is also useful when an actual parameter is a complicated expression and its corresponding formal parameter is referenced more than once in the body of the macro.

The parameter list of a macro invocation can be continued on additional lines, if necessary. A line to be continued must end with a comma that separates two parameters, and the subsequent continuation line must have an ampersand (&) in the first column. If continuation lines are used, a comment can appear only after the final continuation line.

3.8.1.4. Local Labels

If a label is defined within a macro, it must be declared to be local to that macro if the macro will be used more than once in a file; otherwise, multiple invocations of the macro will produce multiple definitions of the label.

Labels local to a macro can be generated by using the **\@** designator as part of the label name or as the label name itself. When the macro is invoked, the **\@** designator will be replaced with a period (.) followed by a unique four-digit number. This number begins at zero and is incremented each time a macro is invoked, thus providing up to 10,000 unique local macro scopes.

3.8.1.5. NARG Symbol

The **NARG** symbol is used in the body of a macro definition. When a macro is expanded, this symbol is replaced with the index of the last parameter with which the macro was invoked (even if this parameter was explicitly specified as null).

The **NARG** symbol is typically used in conjunction with conditional assembly directives to allow a macro's generated code to depend on the number of parameters supplied to it. This symbol has no meaning outside of a macro definition and can be used as an ordinary symbol.

3.8.1.6. MEXIT Directive

The **MEXIT** directive causes the current macro invocation to be terminated; any remaining assembler statements in the macro definition are skipped (i.e., all statements between the **MEXIT** and **ENDM** directives). Only the current invocation is terminated; if macros are nested, control is returned to the previous level of macro expansion.

The **MEXIT** directive is typically used in conjunction with conditional assembly directives to provide a conditional macro termination (see **IFcc** directive).

3.8.1.7. Macro Examples

The remainder of this section provides a series of examples that illustrate the various features of the macro facility. Each example includes a macro definition and one or more invocations of that macro. The **OPT MEX** directive is required to show the macro expansion in the listing file.

The first example presents a macro that can be used to repeat an arbitrary statement a specified number of times. It can be easily modified to accept any number of statements, including invocations of other macros. Following the macro definition is an example invocation and the assembler statements it generates.

```
REP     MACRO
        IFEQ            \2
        MEXIT
        ENDC
        REP             <\1>,?(\2-1)
        \1
        ENDM
        OPT             MEX
        REP             <MOVE.B (A0)+,(A1)+>,8
        MOVE.B          (A0)+,(A1)+
        MOVE.B          (A0)+,(A1)+
        MOVE.B          (A0)+,(A1)+
        MOVE.B          (A0)+,(A1)+
        MOVE.B          (A0)+,(A1)+
        MOVE.B          (A0)+,(A1)+
        MOVE.B          (A0)+,(A1)+
        MOVE.B          (A0)+,(A1)+
```

If the parameter evaluation operator (?) is omitted from the above example, the macro will produce the identical code; however, evaluation of the second parameter will involve redundant computations (e.g., the second parameter would be 8-1-1-1-1-1-1-1-1 — instead of 0 — on the final invocation).

The second example presents a macro that can be used to define sequences of equate symbols; the value of each symbol is a function of its position in the specified sequence. The definition of this macro is followed by a pair of example invocations and their associated expansions.

```
VALUES MACRO
       IFLT          \2
       MEXIT
       ENDC
       VALUES        \1,?(\2-1),\3
\1\2   EQU           \3\2
       ENDM

       OPT           MEX

       VALUES        BIT,7,1<<

BIT0   EQU           1<<0
BIT1   EQU           1<<1
BIT2   EQU           1<<2
BIT3   EQU           1<<3
BIT4   EQU           1<<4
BIT5   EQU           1<<5
BIT6   EQU           1<<6
BIT7   EQU           1<<7

       VALUES        IDX,9,4*

IDX0   EQU           4*0
IDX1   EQU           4*1
IDX2   EQU           4*2
IDX3   EQU           4*3
IDX4   EQU           4*4
IDX5   EQU           4*5
IDX6   EQU           4*6
IDX7   EQU           4*7
IDX8   EQU           4*8
IDX9   EQU           4*9
```

The final example presents a macro that can be used to perform copying of strings and arbitrary memory ranges. Its parameters include the source and destination addresses, the type of copy to be performed, and the number of bytes to be copied (not used for string copies). The final two parameters are optional; they specify the address registers that will be used for the copy. The macro's size qualifier specifies the number of bytes to be copied at a time. The macro **COUNT** uses the qualifier to compute the number of move instructions needed to perform the copy. Two example invocations are shown following the macro definitions.

```
COPY MACRO
       IFNC          '\3','STRING COPY'
       COUNT         \4,\0,MOVEQ,D0
       ENDC
       IFEQ          NARG-6
       LEA           \1,A\5
```

```
        LEA         \2,A\6
L\@
        MOVE.\0     (A\6)+,(A\5)+           ; \3
        ELSEC
        LEA         \1,A0
        LEA         \2,A1
L\@
        MOVE.\0     (A1)+,(A0)+             ; \3
        ENDC
        IFC         '\3','STRING COPY'
        BNE         L\@
        ELSEC
        DBF         D0,L\@
        ENDC
        ENDM

COUNT MACRO
        IFC         '\2','B'
        \3          #(\1-1),\4
        ELSEC
        IFC         '\2','L'
        \3          #((\1>>2)-1),\4
        ELSEC
        \3          #((\1>>1)-1),\4
        ENDC
        ENDC
        ENDM

        OPT         MEX

        COPY.B      MESG1,MESG2,<STRING COPY>,,3,4

        LEA         MESG1,A3
        LEA         MESG2,A4
L.0000
        MOVE.B      (A4)+,(A3)+             ; STRING COPY
        BNE         L.0000

        COPY.L      NODE1,NODE2,<STRUCTURE COPY>,64

        MOVEQ       #((64>>2)-1),D0
        LEA         NODE1,A0
        LEA         NODE2,A1
L.0001
        MOVE.L      (A1)+,(A0)+             ; STRUCTURE COPY
        DBF         D0,L.0001
```

3.8.2. Structured Control Macros

The structured control macro facility provided by **asm68k** is a set of high-level language constructs, which are used to generate run-time loops and conditional execution. These macros expand into the appropriate assembly code to perform the desired control structure. Since these macros are implemented efficiently, they improve readability without sacrificing the desirable aspects of using assembly language. The remainder of this section describes their usage.

> **Note:** Structured control macros do *not* provide assembly-time control — they provide run-time control.

3.8.2.1. Structured Control Expressions

Structured control expressions are used to specify the flow of execution for certain structured control macros. These expressions are translated into one or more **CMP**, **BRA**, and **Bcc** instructions to provide the necessary flow of control. The expressions themselves have a logical value of true of false. They are used with the **IF**, **UNTIL**, and **WHILE** directives. The syntax for constructing structured control expressions is as follows:

Syntax

I. **<*cc1*> [*logical_op*[**.***size*] <*cc2*>]**

II. **<*cc1*> [*logical_op*[**.***size*] *op3* <*cc2*> *op4*]**

III. ***op1*** **<*cc1*> *op2* [*logical_op*[**.***size*] <*cc2*>]**

IV. ***op1*** **<*cc1*> *op2* [*logical_op*[**.***size*] *op3* <*cc2*> *op4*]**

Description

cc1
cc2
Specifies one of the integer conditional tests shown in Table 3.30. Floating-point and PMMU conditional tests are not supported. The angle brackets that enclose the conditional test are required characters.

op1
op2
op3
op4
Specifies an effective address expression (see section **3.5 Effective Addressing Modes**).

logical_op
Specifies one of the following logical operators:

AND	Logical AND
OR	Logical OR

size Specifies the size of the **CMP** instruction generated for the expression that follows the logical operator. (The size qualifier for the first expression is attached to the preceding **IF**, **UNTIL**, or **WHILE** directive.) The legal values are shown below:

B Byte Integer
W Word Integer
L Long-word Integer
S Single-precision Real (not supported)
D Double-precision Real (not supported)
X Extended-precision Real (not supported)
P Packed Decimal Real (not supported)

A **CMP** instruction is generated for each specified pair of effective address operands. If necessary, the operands of this instruction will be exchanged to produce a legal instruction; in this case, the corresponding conditional test will also be reversed. The operands will be exchanged in the following cases:

- The second operand is immediate data.
- The first operand is a data or address register, and the second operand is not a data or address register.

A conditional branch is generated for each specified conditional test. If necessary, the negation of a conditional test will be used to produce the correct flow of control.

Examples

```
<NE>
D1 <GT> #LIMIT
A0 <LT> A1 AND.L D2 <NE> #0
```

Mnemonic	Condition
CC	Carry Clear
CS	Carry Set
EQ	Equal
GE	Greater or Equal
GT	Greater Than
HI	High
LE	Less or Equal

Mnemonic	Condition
LS	Low or Same
LT	Less Than
MI	Minus
NE	Not Equal
PL	Plus
VC	Overflow Clear
VS	Overflow Set

Table 3.30: Integer Conditional Tests

3.8.2.2. Macro Invocation

The structured control constructs are specified with a set of predefined macros, which are shown in Table 3.31. Each of these macros and their associated parameters expand into the appropriate labels and instructions to produce the desired control structure. These expansions will appear in the listing file if the **OPT SCEX** directive is specified (default: **OPT NOSCEX**). An alphabetically arranged set of macro descriptions is provided below. Structured control macros can be nested to form more complicated control structures.

Mnemonic	Function
BREAK	Terminate Loop Execution
CONTINUE	Begin Next Loop Iteration
FOR . . . ENDF	Loop Based on Counter
IF . . . ELSE . . . ENDI	Perform Conditional Execution
REPEAT . . . UNTIL	Loop Until Condition True
WHILE . . . ENDW	Loop While Condition True

Table 3.31: Structured Control Macros

In addition to the macro mnemonics, there is a set of supporting keywords, which are not reserved; they are **AND**, **BY**, **DO**, **DOWNTO**, **OR**, **THEN**, and **TO**. These keywords are required syntax for macro invocations, but can be used elsewhere as ordinary symbols.

As with user-defined macros, invocations of structured control macros can be continued on additional lines, if necessary. Because of the syntax, no continuation characters are needed; however, blank lines cannot appear between any of the continuation lines. Comments used prior to a continuation line must begin with an exclamation point (!).

Note: The labels that are created during the expansion of structured control macros have local scope. They are defined using the same mechanism that is used for labels local to user-defined macros (see section **3.8.1.4 Local Labels**). The labels are named **ZL1\@**, **ZL2\@**, **ZL3\@**, and **ZL4\@**; the **\@** designator is replaced by a period (.) and a unique four-digit number. These labels are treated the same as compiler local labels as far as the symbol tables in the object and listing files are concerned (i.e., their presence or absence is controlled with the **-S** and **-s** command line flags).

3.8.2.3. Structured Control Reference

The remainder of this section contains detailed descriptions of the various structured control macros. The descriptions are ordered alphabetically.

BREAK — Terminate Loop Execution

Syntax

BREAK[.*extent*]

Description

extent Specifies the *extent* of the generated forward branch. The legal values are shown below; if omitted, the branch extent is determined by the current default forward branch size (see **OPT BRB/BRS/BRW** directive).

- **B** 8-bit forward branch
- **S** 8-bit forward branch
- **W** 16-bit forward branch

The **BREAK** macro is analogous to the **break** statement in C — i.e., it terminates execution of the smallest enclosing loop. It is used to exit a **FOR**, **REPEAT**, or **WHILE** loop before the loop's termination condition is met.

The **BREAK** macro expands into an unconditional branch to the assembler-generated label that immediately follows the smallest enclosing loop. Using this macro outside a loop results in an error.

CONTINUE — Begin Next Loop Iteration

Syntax

CONTINUE[.*extent*]

Description

extent Specifies the *extent* of the generated forward branch. The legal values are shown below; if omitted, the branch extent is determined by the current default forward branch size (see **OPT BRB/BRS/BRW** directive).

- **B** 8-bit forward branch
- **S** 8-bit forward branch
- **W** 16-bit forward branch

The **CONTINUE** macro is analogous to the **continue** statement in C — i.e., it proceeds to the next iteration of the smallest enclosing loop. It is used to skip the remainder of the current iteration of a **FOR**, **REPEAT**, or **WHILE** loop.

The **CONTINUE** macro expands into an unconditional branch to the assembler-generated label that immediately precedes the condition test of the smallest enclosing loop. Using this macro outside a loop results in an error.

FOR . . . ENDF — Loop Based on Counter

Syntax

FOR[.*size*] *op1* = *op2* { TO | DOWNTO } *op3* [BY *op4*] DO[.*extent*]
[*statement*]
.
.
.
ENDF

Description

size Specifies the size qualifier for instructions that are generated to operate on *op1*, *op2*, *op3*, and *op4*. The legal values are shown below; if omitted, the size is determined according to the rules discussed in section **3.4.2 Instruction Sizing**.

- **B** Byte Integer
- **W** Word Integer
- **L** Long-word Integer

op1 Specifies the loop counter. It must be an alterable effective address expression (see section **3.5 Effective Addressing Modes**).

op2 Specifies the initial value of the loop counter. It can be any effective address expression.

op3 Specifies the final value of the loop counter. It can be any effective address expression.

op4 Specifies the step value (increment/decrement) for the loop counter. It can be any effective address expression. If omitted, it defaults to #1.

extent Specifies the *extent* of the forward branch that is generated to span the loop body. The legal values are shown below; if omitted, the branch extent is determined by the current default forward branch size (see **OPT BRB/BRS/BRW** directive).

- **B** 8-bit forward branch
- **S** 8-bit forward branch
- **W** 16-bit forward branch

statement Specifies an assembler statement.

The **FOR . . . ENDF** macro is a restricted form of the **for** statement in C. It generates a counter-based iterated loop, which can be executed zero or more times. The loop is executed until *op1* is greater / less (**TO / DOWNTO**) than *op3*.

This macro uses a **MOVE** instruction to initialize *op1* to the value of *op2*, either an **ADD** instruction to increment *op1* by *op4* (**TO**) or a **SUB** instruction to decrement *op1* by *op4* (**DOWNTO**), and a **CMP** instruction to perform the termination test involving *op1* and *op3*. An unconditional forward branch is used initially to jump to the termination test, which is situated at the end of the loop; a conditional backward branch is used to jump to the beginning of the loop to perform the next iteration. The initial jump to the termination test will be omitted during expansion if it can be determined that the loop will execute at least once.

Upon normal exit from the loop, *op1* will contain the value that caused the loop to terminate; the condition codes will reflect the final execution of the **CMP** instruction. Since the **FOR . . . ENDF** construct is simply a macro, there are no restrictions on modifying the operands *op1*, *op2*, *op3*, and *op4* from within the body of the loop.

Examples

```
FOR.L A0 = #OUTBUF TO #OUTBUF+BUFSIZ-1 BY #4 DO.S
    CLR.L (A0)
ENDF

FOR.W D0 = #1 TO #SIZE DO.S
    MOVE.B (A0)+,(A1)+
ENDF
```

As the first example illustrates, the default step size of 1 is inappropriate when the loop counter is used directly to index through word or long-word data.

IF . . . ELSE . . . ENDI — Perform Conditional Execution

Syntax

IF[.*size*] *expression* THEN[.*extent*]
[*statement*]
.
.
.
ENDI

– or –

IF[.*size*] *expression* THEN[.*extent*]
[*statement*]
.
.
.
ELSE[.*extent*]
[*statement*]
.
.
.
ENDI

Description

size Specifies the size qualifier for the structured control *expression*. The legal values are shown below; if omitted, the size is determined according to the rules discussed in section **3.4.2 Instruction Sizing**.

- **B** Byte Integer.
- **W** Word Integer.
- **L** Long-word Integer.
- **S** Single-precision Real (not supported).
- **D** Double-precision Real (not supported).
- **X** Extended-precision Real (not supported).
- **P** Packed Decimal Real (not supported).

expression Specifies a structured control expression (see section **3.8.2.1 Structured Control Expressions**).

extent Specifies the *extent* of the forward branches that are generated to span the **THEN** and **ELSE** statement groups. The legal values are shown below; if omitted, the branch extent is determined by the current default forward branch size (see **OPT BRB/BRS/BRW** directive).

B 8-bit forward branch.

S 8-bit forward branch.

W 16-bit forward branch.

statement Specifies an assembler statement.

The **IF . . . ELSE . . . ENDI** macro is analogous to the **if-else** construct in C. If the specified expression is true, the set of statements following the **THEN** keyword is executed; otherwise, the set of statements following the **ELSE** keyword (if present) is executed.

Code is generated to evaluate the structured control expression *expression* and to perform the necessary flow of control. This will include either one or two branches, depending on the expression's complexity.

When **IF** macros are nested, each **ELSE** clause is associated with the immediately preceding **IF** macro that is not matched by an **ENDI** or **ELSE** directive.

Example

```
IF.W D1 <GT> D2 THEN.S
   MOVE.W D1,D0
ELSE.S
   MOVE.W D2,D0
ENDI
```

REPEAT . . . UNTIL — Loop Until Condition True

Syntax

REPEAT
    [*statement*]
    .
    .
    .
UNTIL[*.size*] *expression*

Description

statement Specifies an assembler statement.

size Specifies the size qualifier for the structured control *expression*. The legal values are shown below; if omitted, the size is determined according to the rules discussed in section **3.4.2 Instruction Sizing**.

- **B** Byte Integer
- **W** Word Integer
- **L** Long-word Integer

expression Specifies a structured control expression (see section **3.8.2.1 Structured Control Expressions**).

The **REPEAT . . . UNTIL** macro generates a loop that executes until the specified condition becomes true. The termination test is performed at the end of the loop; therefore, the loop is executed at least once, even if the terminating condition is true upon entry.

Code is generated to evaluate the structured control expression *expression* and to perform the necessary flow of control. This will include either one or two branches, depending on the expression's complexity. Upon normal exit from the loop, the condition codes will reflect the final evaluation of *expression*.

To repeat an instruction or sequence of instructions at assembly-time, refer to the **REP** macro in the first example in section **3.8.1.7 Macro Examples**.

Example

```
REPEAT
    MOVE.B (A0)+,(A1)+
UNTIL <EQ>
```

WHILE . . . ENDW — Loop While Condition True

Syntax

WHILE[.*size*] *expression* DO[.*extent*]
[*statement*]
.
.
.
ENDW

Description

size Specifies the size qualifier for the structured control *expression*. The legal values are shown below; if omitted, the size is determined according to the rules discussed in section **3.4.2 Instruction Sizing**.

- **B** Byte Integer
- **W** Word Integer
- **L** Long-word Integer

expression Specifies a structured control expression (see section **3.8.2.1 Structured Control Expressions**).

extent Specifies the size of any forward branches that are generated to span the loop body. The legal values are shown below; if omitted, the size is determined by the current default forward branch size (see **OPT BRB/BRS/BRW** directive).

- **B** 8-bit forward branch
- **S** 8-bit forward branch
- **W** 16-bit forward branch

statement Specifies an assembler statement.

The **WHILE . . . ENDW** macro is analogous to the **while** statement in C — i.e., it generates a loop that executes while a specified condition is true. The termination test is performed at the beginning of the loop; therefore, the loop is not executed if the condition is false upon entry.

Code is generated to evaluate the structured control expression *expression* and to perform the necessary flow of control. This will include either one or two forward branches, depending on the expression's complexity. An unconditional backward branch is used to perform the next iteration of the loop. Upon normal exit from the loop, the condition codes will reflect the final evaluation of *expression*.

Example

```
WHILE.L (A0) <NE> #-1 DO.S
    ADD.L (A0)+,D0
ENDW
```

3.9. Instruction Set Summary

The instructions for the 68000 family of microprocessors and coprocessors are listed alphabetically in Table 3.32. However, only the 68000 instruction set is supported by Texas Instruments. Use of unsupported instructions gives unpredictable results. This is especially true of the floating-point instructions. The legal size qualifiers and the default sizes are summarized for each instruction.

Mnemonic	Size Qualifiers	Default Size		Not Supported
		asm68	asm68k	
ABCD	B	B	B	
ADD	B W L	L	W	
ADDA	W L	L	W	
ADDI	B W L	L	W	
ADDQ	B W L	L	W	
ADDX	B W L	L	W	
AND	B W L	L	W	
ANDI	B W L	L	W	
ANDI to CCR	B	B	B	
ANDI to SR	W	W	W	
ASL	B W L	L	W	
ASR	B W L	L	W	
Bcc	B W L†	W	W	
BCHG	B L	B/L‡	B/L‡	
BCLR	B L	B/L‡	B/L‡	
BFCHG				✓
BFCLR				✓
BFEXTS				✓
BFEXTU				✓
BFFFO				✓
BFINS				✓
BFSET				✓
BFTST				✓
BGND				✓
BKPT				✓
BRA	B W L†	W	W	
BSET	B L	B/L‡	B/L‡	
BSR	B W L†	W	W	
BTST	B L	B/L‡	B/L‡	
CALLM				✓
CAS	B W L	L	W	✓
CAS2	W L	L	W	✓
CHK	W L†	L	W	
CHK2	B W L	L	W	✓

Table 3.32: Instructions and Size Qualifiers

Mnemonic	Size Qualifiers	Default Size		Not Supported
		asm68	asm68k	
CINV				✓
CLR	B W L	L	W	
CMP	B W L	L	W	
CMP2	B W L	L	W	✓
CMPA	W L	L	W	
CMPI	B W L	L	W	
CMPM	B W L	L	W	
cpBcc	W L	W	W	✓
cpDBcc	W	W	W	✓
cpGEN				✓
cpRESTORE				✓
cpSAVE				✓
cpScc	B	B	B	✓
cpTRAPcc	W L	W	W	✓
CPUSH				✓
DBcc	W	W	W	
DIVS	W L†	L	W	
DIVSL	L	L	L	✓
DIVU	W L†	L	W	
DIVUL	L	L	L	✓
EOR	B W L	L	W	
EORI	B W L	L	W	
EORI to CCR	B	B	B	
EORI to SR	W	W	W	
EXG	L	L	L	
EXT	W L	L	W	
EXTB	L	L	L	✓
FABS	B W L S D X P	X	W	✓
FACOS	B W L S D X P	X	W	✓
FADD	B W L S D X P	X	W	✓
FASIN	B W L S D X P	X	W	✓
FATAN	B W L S D X P	X	W	✓
FATANH	B W L S D X P	X	W	✓
FBcc	W L	W	W	✓

Table 3.32: Instructions and Size Qualifiers (continued)

Mnemonic	Size Qualifiers	Default Size		Not Supported
		asm68	asm68k	
FCMP	B W L S D X P	X	W	✓
FCOS	B W L S D X P	X	W	✓
FCOSH	B W L S D X P	X	W	✓
FDABS	B W L S D X P	X	W	✓
FDADD	B W L S D X P	X	W	✓
FDBcc				✓
FDDIV	B W L S D X P	X	W	✓
FDIV	B W L S D X P	X	W	✓
FDMOVE	B W L S D X P	X	W	✓
FDMUL	B W L S D X P	X	W	✓
FDNEG	B W L S D X P	X	W	✓
FDSQRT	B W L S D X P	X	W	✓
FDSUB	B W L S D X P	X	W	✓
FETOX	B W L S D X P	X	W	✓
FETOXM1	B W L S D X P	X	W	✓
FGETEXP	B W L S D X P	X	W	✓
FGETMAN	B W L S D X P	X	W	✓
FINT	B W L S D X P	X	W	✓
FINTRZ	B W L S D X P	X	W	✓
FLOG10	B W L S D X P	X	W	✓
FLOG2	B W L S D X P	X	W	✓
FLOGN	B W L S D X P	X	W	✓
FLOGNP1	B W L S D X P	X	W	✓
FMOD	B W L S D X P	X	W	✓
FMOVE	B W L S D X P	X	W	✓
FMOVECR	X	X	X	✓
FMOVEM	L X	X	X	✓
FMUL	B W L S D X P	X	W	✓
FNEG	B W L S D X P	X	W	✓
FNOP				✓
FREM	B W L S D X P	X	W	✓
FRESTORE				✓
FSABS	B W L S D X P	X	W	✓
FSADD	B W L S D X P	X	W	✓
FSAVE				✓

Table 3.32: Instructions and Size Qualifiers (continued)

Mnemonic	Size Qualifiers	Default Size		Not Supported
		asm68	asm68k	
FSCALE	B W L S D X P	X	W	✓
FScc	B	B	B	✓
FSDIV	B W L S D X P	X	W	✓
FSGLDIV	B W L S D X P	X	W	✓
FSGLMUL	B W L S D X P	X	W	✓
FSIN	B W L S D X P	X	W	✓
FSINCOS	B W L S D X P	X	W	✓
FSINH	B W L S D X P	X	W	✓
FSMOVE	B W L S D X P	X	W	✓
FSMUL	B W L S D X P	X	W	✓
FSNEG	B W L S D X P	X	W	✓
FSQRT	B W L S D X P	X	W	✓
FSSQRT	B W L S D X P	X	W	✓
FSSUB	B W L S D X P	X	W	✓
FSUB	B W L S D X P	X	W	✓
FTAN	B W L S D X P	X	W	✓
FTANH	B W L S D X P	X	W	✓
FTENTOX	B W L S D X P	X	W	✓
FTRAPcc	W L	W	W	✓
FTST	B W L S D X P	X	W	✓
FTWOTOX	B W L S D X P	X	W	✓
ILLEGAL				
JMP				
JSR				
LEA	L	L	L	
LINK	W L†	W	W	
LPSTOP	W	W	W	✓
LSL	B W L	L	W	
LSR	B W L	L	W	
MOVE	B W L	L	W	
MOVE from CCR	W	W	W	✓
MOVE from SR	W	W	W	
MOVE to CCR	W	W	W	
MOVE to SR	W	W	W	
MOVE USP	L	L	L	

Table 3.32: Instructions and Size Qualifiers (continued)

Mnemonic	Size Qualifiers	Default Size		Not Supported
		asm68	asm68k	
MOVE16	L	L	L	✓
MOVEA	W L	L	W	
MOVEC	L	L	L	✓
MOVEM	W L	L	W	
MOVEP	W L	L	W	
MOVEQ	L	L	L	
MOVES	B W L	L	W	✓
MULS	W L†	L	W	
MULU	W L†	L	W	
NBCD	B	B	B	
NEG	B W L	L	W	
NEGX	B W L	L	W	
NOP				
NOT	B W L	L	W	
OR	B W L	L	W	
ORI	B W L	L	W	
ORI to CCR	B	B	B	
ORI to SR	W	W	W	
PACK				✓
PBcc	W L	W	W	✓
PDBcc	W	W	W	✓
PEA	L	L	L	
PFLUSH				✓
PFLUSHA				✓
PFLUSHR				✓
PFLUSHS				✓
PLOAD				✓
PMOVE	B W L D	D	D	✓
PRESTORE				✓
PSAVE				✓
PScc	B	B	B	✓
PTEST				✓
PTRAPcc	W L	L	W	✓
PVALID	L	L	L	✓
RESET				

Table 3.32: Instructions and Size Qualifiers (continued)

Mnemonic	Size Qualifiers	Default Size		Not Supported
		asm68	asm68k	
ROL	B W L	L	W	
ROR	B W L	L	W	
ROXL	B W L	L	W	
ROXR	B W L	L	W	
RTD				✓
RTE				
RTM				✓
RTR				
RTS				
SBCD	B	B	B	
Scc	B	B	B	
STOP				
SUB	B W L	L	W	
SUBA	W L	L	W	
SUBI	B W L	L	W	
SUBQ	B W L	L	W	
SUBX	B W L	L	W	
SWAP	W	W	W	
TAS	B	B	B	
TBLS	B W L	L	W	✓
TBLSN	B W L	L	W	✓
TBLU	B W L	L	W	✓
TBLUN	B W L	L	W	✓
TRAP				
TRAPcc	W L	L	W	✓
TRAPV				
TST	B W L	L	W	
UNLK				
UNPK				✓

† Size qualifier .L legal with 68020/30/40 and CPU32 only
‡ Size qualifier is determined by the destination operand: .B (memory), .L (register)

Table 3.32: Instructions and Size Qualifiers (continued)

Section 4: Linker

4. Linker

4.1. Introduction

The Sierra Systems linker **link68** combines object files to create an executable output file. It was developed by Sierra Systems to support certain Motorola processors in conjunction with the rest of the Sierra C™ software package. Under license from Sierra Systems, Texas Instruments has modified this software to support TI BCD floating-point numbers, and support for coprocessors has been removed. Although the software has not been modified to exclude support for processors other than the 68000, the 68000 is the only processor supported by Texas Instruments. The license from Texas Instruments to use these products is restricted to development of software that is targeted to execute only on TI calculators.

4.2. Link68 Inputs and Outputs

The linker's actions are directed by the arguments placed on the command line. The linker opens each file referenced in its command input and reads the file header to determine if the file is an object or library file. If the file is an object file, the linker resolves undefined addresses and concatenates each section in the file to an output section of the same name. Otherwise, if the file is a library, the linker searches the symbol table of the library file to determine if any undefined symbols may be defined by an object file in the library. If the linker finds such a symbol in the library, it loads the member defining that symbol from the library. The search continues until no undefined symbols can be resolved by loading library members. Only the library members needed to resolve undefined symbols are loaded. The linker treats the object files loaded from a library as if the names of the object files themselves appeared in the command input. If the file is neither an object nor library file, the linker generates an error message and aborts.

The linker's primary output is an object file that, by default, is executable. The linker copies code from input object files named in its command input to the executable file, modifying the code to reflect its address in the executable. Information needed for further linking, such as relocation entries, is removed from the executable file.

If the command input includes the `-m` flag specifying a map file, the linker will create a formatted listing of the executable file's contents, showing where each input file is located in the generated output file. The map file's name is simply the executable file's base name with a **.map** extension.

4.3. Options

This section describes the various library search options and command line flags available for applications developers.

Typically, the TI **FLASH** Studio™ will handle all invocations of the linker, using the correct command line flags required to produce TI-89 / TI-92 Plus apps or ASMs. The following discussion of command line format and flags is included for developers who may wish to use **link68** directly from the command line or create their own makefile.

4.3.1. Library Search Options

The linker provides the **-l** flag for loading standard library files. The linker searches for the specified library in the directories specified by the environment (e.g., **sierra/lib**) and the **-L** flag. The library name searched for is created by prefixing the string that follows the **-l** flag with **lib** and suffixing it with **.68**. For example, given **-l xyz** the library **libxyz.68** will be searched for. The linker examines each directory on its list of standard library directories and if it finds the library in one of them, it stops searching and processes the library as already described. Note that the **-l** flag is not required to specify a library file, the full pathname of the library can also be specified.

The **-L** flag adds a directory to the linker's library search path (used with the **-l** flag). The directories specified with the **-L** flag are searched in the order in which they appear in the command input. The remaining directories in the library search path are obtained from the program's environment as follows. If the environment variable **LIB68** is defined, each of the directories it specifies is added to the search path. Otherwise, if the **SIERRA** environment variable is defined, the directory it specifies (suffixed with **/lib**) is added to the search path. The directories specified with the **-L** flag are searched before those specified by the environment.

Texas Instruments provides only one library file, **libams.68**. Many functions previously included in the Sierra C library files have been rewritten as part of the calculator operating system and are accessed through the jump table instead. No other library file is required to create apps or ASMs for use on the TI-89 or TI-92 Plus.

4.3.2. Option Flags

A sample invocation of the linker is included with the files supplied with the TI-89 / TI-92 Plus SDK. It is strongly recommended that you use the linker only as shown in those examples. The following option flags can also be included on the command line if desired. Whitespace is optional between a flag and any

argument. The linker accepts multiple flag letters grouped after a single hyphen (-) provided only the last flag can take an argument.

-E Cause undefined symbols to be an error instead of a warning. Errors cause the linker to exit (after completion) with a nonzero return value.

-i *file* Include (interpolate) the file *file* at the current position in the command line or command file. Command files may be nested inside one other.

-h *hci_file* Generate hole compression information for the linker input files listed in the file *hci_file*. The *hci_file* is the hole compression file created by **asm68** using the **asm68 -h** flag. The resultant hole compression information is written to a file with a name created by stripping the **.hci** extension (if present) from the *hci_file* and appending **.hco**.

-H *hco_file* Determine if any compressed holes are too small. If any holes are incorrectly compressed, make corrections in the *hco_file*, the hole compression output file generated during the first linker pass in response to the **-h** flag. If an additional assembly-linkage pass is required (unlikely), a message is generated indicating that the additional pass is necessary. The **-H** flag must be used on the second hole compression linker pass and on any subsequent passes.

-l *libspc* Search for the specified library in the library directories specified by the environment (i.e., **sierra/lib**) and the **-L** flag. The library name searched for is created by prefixing *libspc* with **lib** and suffixing it with **.68**. For example, given **-lms** the library **libms.68** will be searched for. The **-x** flag can be used to change the default **.68** suffix.

-L *dir* Add the directory pathname *dir* to the library paths to be searched when the **-l** flag is specified. The **-L** flag can be specified multiple times.

-m Generate a map file. The name of the map file is created by stripping the **.out** extension, if present, from the output file's name and appending **.map**.

-o *outfile* Name the output file *outfile*. If the **-o** flag is not used to specify an output filename, the name **x.out** is used.

-Q Suppress the Sierra Systems copyright message. This flag must appear before any filenames or linker directives to be effective.

-r Build relocation information for the app.

-S Silence all warning messages.

-u Print usage information.

-x *ext* Set the extension for standard libraries specified with the **-l** flag to *.ext*. The extension *ext* can be up to three characters long. The default extension is **.68**.

4.4. Object Files

Object files are generated by the assemblers (**asm68**, **asm68k**) and the linker (**link68**). Object files include three major components: sections, relocation information, and a symbol table. The sections component primarily stores the text and data of a program. The relocation information describes where and how the code in the various text and data sections must be modified as the linker builds the executable output file. The symbol table holds symbols (labels) together with associated addresses and absolute values.

4.4.1. Sections

Sections are indivisible, contiguous pieces of object code. When creating an app, sections must be defined and used only as shown in the examples supplied with the TI-89 / TI-92 Plus SDK. See chapter **7. Flash Application Layout** in the TI-89 / TI-92 Plus Developers Guide for more information.

4.5. Symbols

Symbols are address labels and assignment variables, each of which can be either defined or undefined. If a symbol refers to an address in a relocatable section, it is called *relocatable* because its value changes as the linker relocates its section. If a symbol refers to a numeric constant or an address in an absolute section, it is called *absolute* because its value cannot be changed. Relocatable and absolute symbols are referred to as *defined*. The values associated with defined symbols are stored as 32-bit numbers. Symbols that are referenced but not defined are referred to as *undefined*. An undefined symbol's value and type are resolved by the linker.

A symbol is also either *global* or *local*. The linker uses only global symbols to resolve undefined symbols and to search libraries. Local symbols are kept in object files purely for informational and debugging purposes. All undefined symbols are global.

A variant of the undefined symbol type is the comm symbol. The compiler generates a comm symbol whenever it encounters a declaration for an uninitialized external variable. A comm symbol is always global and the linker defines an uninitialized block of data in the **.bss** section for it. The length of the block and its alignment are encoded in the 32-bit value field associated with the comm symbol. For information on how the length and alignment of comm symbols are encoded, refer to section **1.4.8.5 Section Number Field**.

4.6. Relocation Entries

An object file contains relocation information that describes how the code in the file must be modified when it is copied to the executable output file. The information is stored as a series of relocation entries. Each relocation entry locates a byte, word, or long word in the object code that refers to an undefined symbol. The piece of code referenced by a relocation entry is called a *relocation hole*. A relocation entry also specifies a symbol that the linker adds to the hole when the linker moves code or resolves undefined symbols.

4.7. Relocation Hole Compression

When the linker fills in a relocation hole, it is possible that the value it places there could have fit in a smaller hole than was left by the assembler. This is because the assembler, unless otherwise directed, leaves 32-bit absolute relocation holes for references it cannot resolve. To avoid wasting space, **asm68** and **link68** together provide an option called *relocation hole compression* which can be used to shrink oversized holes. Shrinking relocation holes not only reduces the size of your program, but also increases its execution speed.

The hole compression technique used by Sierra Systems uses a double assembly and linkage pass, and decreases code size by an average of four to seven percent. The actual size decrease (speed increase) depends on a number of factors, such as the placement of the various output sections and the number of external references in those sections. The increase in time resulting from the double assembly/linkage passes has been determined to be minimal, especially when considering that the compression process will typically be performed only once as a final optimization. An example of both invocations of the linker required for hole compression is shown below. The majority of the linker directives are contained in the file app.lnk in this example. Refer to the sample files supplied with the TI-89 / TI-92 Plus SDK for information on what should be included in app.lnk.

```
link68 -h holes -i app.lnk

link68 -r -H holes.hco -i app.lnk
```

During the first pass, the assembler and linker are each invoked as they ordinarily are, with the addition of the **-h** flag. The argument to this flag specifies a file that will contain the names of the object files on which hole compression is to be performed. It is generated incrementally by the assembler during each of its first pass invocations and is used as an input file to the linker. This file is referred to as the hole compression input file, and by convention is designated by the file extension **.hci**. In addition to the names of the object files, this file includes the command line arguments from each assembler invocation (needed for regenerating the object files), as well as the size of the target processor's address bus (needed to ensure correct address computations by the linker).

Once all the assembly files have been processed by the assembler, the linker is invoked with the **-h** flag with the hole compression input file just created specified as an argument. For each 32-bit relocation hole in each TEXT-like section of each object file listed in this file, the linker determines whether the address of the referenced symbol could be expressed as an absolute short address and/or a 16-bit program counter relative address. This information is stored along with the address of the relocation hole in the compression output file. The name of this file is created by stripping the **.hci** extension, if present, from the compression input file, and appending **.hco**. Also stored in this file are the address bus size and the assembler command line arguments, both of which are obtained from the compression input file. Once the compression input file is no longer needed, it is moved to a backup file which is designated by the file extension **.hcb**.

During the second pass, the assembler and linker are each invoked with the **-H** flag, followed by the name of the hole compression output file generated by the linker. The assembler treats the compression information contained in this file as suggestions only, since it is possible that the addressing mode required to compress a hole will not be legal for the effective address field of a given instruction. If the addressing mode required to compress a hole is legal, though, the assembler will shrink the hole to 16 bits, opting for the PC-relative mode over the absolute short mode if both addressing modes are legal and permit compression. The assembler requires only a single invocation during this pass, because it reads the command lines needed to assemble each input file directly from the compression output file; no command line arguments beside the **-H** flag and its file should be specified during this pass.

Finally, the linker is invoked with the **-H** flag to link the compressed object files. It is possible, albeit unlikely, that there will exist a compressed relocation hole that is too small for the value it must hold. This can only occur when other hole compressions have caused an increase in the distance between the hole and the location it references. If this has occurred, the linker will edit the hole compression output file to rescind the compression recommendation for that hole. An additional pass (identical to the second pass) will then be required to uncompress that hole and any others like it. A message will notify the user if an extra pass is necessary. Only in the rarest of cases will more than one additional pass be required. If there is still an error after the second pass, check your source code for possible problems.

The two assembly/linkage passes (plus any necessary additional passes) required for relocation hole compression are performed automatically by the TI **FLASH** Studio.

4.8. Reserved Symbols

The Sierra Systems standard C libraries reserve several symbols to locate the memory heap. The symbols **heap_org** and **heap_len** and the memory routines that use them, such as **malloc**, are not supported by Texas Instruments. However, they are still reserved names and must be avoided. See chapter **13. Memory Management** in the TI-89 / TI-92 Plus Developers Guide for information on the memory routines available on the TI-89 / TI-92 Plus.

Section 5: Utilities

5. Utilities

5.1. Symbol Table Name Utility

Name

nm68 – Sierra Systems Symbol Table Name Utility

Syntax

nm68 `[-abefhnrstuvFHRTU] [-d[#[:#|+#]]] [`*`files . . .`*` ]`

Description

The **nm68** utility displays the symbol table embedded in a 68000 COFF object file, as well as information on the file header, section headers, relocation entries, and line number entries. For detailed information on the Sierra Systems object module format, see section **1.4 Object File Format**. The zero or more files *files* specified on the command line can be either relocatable or absolute COFF object files or libraries of relocatable or absolute object files. When a library file is specified, the member object files are displayed as if they had been listed on the command line individually. If no file name is specified, the file **x.out** is assumed. If a file name without an extension is specified, the extension **.out** is assumed if the file cannot first be found without the extension.

By default, the output from **nm68** contains detailed information about each symbol, including the symbol's name, value (address), storage class, type, size, associated C language source file line, and section. If the original source file was not compiled with the debug option enabled, some of the symbol information may not be available. All values except for addresses are in decimal. Addresses appearing in the value field are in hexadecimal (padded with leading 0's). If the symbol is a comm variable, both the size and alignment requirements of the comm variable appear in the value field; the size of the comm appears to the left of the decimal point and the boundary alignment appears to the right.

Command line flags are:

-a Print the symbol name and its value in the form of an assignment statement such that the generated list can be inserted directly into a C or assembly language source file or a linker command file.

-b Suppress the header information at the start of the formatted and raw symbol listings.

-d [*start_offset*[:*end_offset* | +*length*]]

Dump the contents of the file in both hexadecimal and ASCII starting at file offset *start_offset* and continuing through file offset *end_offset* or the specified *length*. If only *start_offset* is specified, the entire file from that offset to the end is dumped. If the **-d** flag is used without any arguments, the entire file is dumped.

-e Print only external symbols. When used with the **-s** flag, static symbols are also printed.

-f Produce a full listing that includes the symbols **.text**, **.data** and **.bss**, which are normally suppressed.

-F Prefix the name of the object file or library containing the object file to every output line.

-h Print the file header, the optional **a.out** header, and all the individual section headers contained in the object file. In general, indexes, counts, and line numbers are in decimal while sizes, addresses, flags, and file offsets (enclosed in parentheses) are in hexadecimal. The file header listing includes the magic number, date, number of sections, flags, and number of symbols. The number of symbols is followed by the file offsets of the symbol table and string table in parentheses. The optional **a.out** header listing includes a magic number, a version number, an execution entry point, and the sizes and addresses of the **.text**, **.data**, and **.bss** sections. Each section header listing includes both the section's physical and virtual addresses, the section's size followed by the file offset to the start of the section, flags, and the number of associated line numbers and relocation entries together with file offsets. See the example screen display shown on page 312.

-H Print relocation and line number information in addition to the header information printed by the **-h** flag. For both relocation and line number entries, the first column contains the file offset of the entry in hexadecimal.

-n Sort symbols alphabetically by name. Only external symbols are sorted if local symbols are present; if local symbols are suppressed by the **-s**, **-e**, or **-U** flags, static and external symbols are sorted together.

-r Reverse the order of the sort specified by the **-n** or **-v** flags.

-R Print a raw symbol table dump that displays each entry in both ASCII and hexadecimal. The symbol table index in decimal is shown on the left side. The string table entries, together with hexadecimal offsets from the start of the string table, are printed at the end. All symbols are listed in the order that they appear in their respective object files.

-s Print only static symbols. When used with the **-e** flag, external symbols are also printed.

-t Print the symbol information in a terse format that provides each symbol's name, address, and type. The type is indicated by a single letter defined as follows:

- **a** absolute symbol, global
- **A** absolute symbol, local
- **b** bss symbol, local
- **B** bss symbol, global
- **C** comm variable
- **d** data symbol, local
- **D** data symbol, global
- **f** file name
- **r** register variable
- **s** symbol from user defined section, local
- **S** symbol from user defined section, global
- **t** text symbol, local
- **T** text symbol, global
- **U** undefined symbol

-T Do not truncate symbol names and type information to fit in the column space provided. If a name or type overflows the allotted space, the columns to the right of the name or type will be shifted over.

-u Print usage information.

-U Print only undefined external symbols.

-v Sort symbols by value in ascending order. The same sorting restrictions that apply to the **-n** flag also apply to the **-v** flag.

Examples

The following command dumps the symbol table from the object file **hello.o**:

```
nm68 hello.o
```

The output from the above invocation of **nm68** is as follows:

```
Name      Value      Class     Type       Size      Line     Section

hello.c   |          |file     |          |         |        |
_main     |00000064  |extern   |  int()   |    82   |        |.text
.bf       |00000068  |func     |          |         |     2  |.text
i         |      -4  |auto     |   int    |         |        |(ABS)
.ef       |000000b2  |func     |          |         |    10  |.text
_printf   |       0  |extern   |          |         |        |
_scanf    |       0  |extern   |          |         |        |
```

Note: Usually the output from **nm68** will be too large to fit on a single screen. You will probably want to redirect the output to a scratch file, using the redirection operator (>), and then examine the display using a text editor.

The following command dumps the contents of the file and section headers that are part of the executable file **hello.out**:

```
nm68 -h hello.out
```

The output from the above invocation of **nm68** is listed below:

```
MAGIC_NUMBER: 0x0150①                                  Wed Oct 21 01:20:18 1992②

SECTIONS: 4③  SYMBOLS: 216④ (001932⑤, 002862⑥)  FLAGS: 0x0003⑦
A.OUT_MAGIC: 0x0107                                              VERSION: 300

TSIZE:  0017dc⑧    DSIZE:  000008⑧    BSIZE: 0000e0⑧   ENTRY: 005000⑨
TSTART: 005000⑩    DSTART: 006824⑩

.text❶     ADDRESS: p-005000❷ v-005000❸ SIZE: 0017dc❹          FLAGS: 0120❻
           (0000d0❺)
           LINE_ENTRIES:  9❼       (0018fc❽)
           RELOC_ENTRIES: 0❾       (000000⑩)

.ld_tbl    ADDRESS: p-0067dc v-0067dc  SIZE: 000048 (0018ac)     FLAGS: 0120
           LINE_ENTRIES:  0        (001932)
           RELOC_ENTRIES: 0        (000000)

.data      ADDRESS: p-006824 v-006824  SIZE: 000008 (0018f4)     FLAGS: 0140
           LINE_ENTRIES:  0        (001932)
           RELOC_ENTRIES: 0        (000000)

.bss       ADDRESS: p-00682c v-00682c  SIZE: 0000e0 (000000)     FLAGS: 0180
           LINE_ENTRIES:  0        (000000)
           RELOC_ENTRIES: 0        (000000)
```

Key:

① All 68000 family executable files begin with 0x150. See section **1.4.2.1. Magic Number**.
② Date the object file was created.
③ Number of output sections in the object file.
④ Number of symbols in the file.
⑤ File offset (hexadecimal) to the start of the symbol table.
⑥ File offset (hexadecimal) to the start of the string table.
⑦ File header flags. See section **1.4.2.3 Flags**, Table 1.4.
⑧ Size in bytes (hexadecimal) of sections **.text**, **.data** and **.bss**.
⑨ Entry point or starting address (hexadecimal) of program.
⑩ Starting address (hexadecimal) of sections **.text** and **.data**.
❶ Section name. Empty if section is an unnamed **ORG** section.
❷ Physical address (hexadecimal) of section. See section **1.4.1.2 Physical and Virtual Addresses**.
❸ Virtual address (hexadecimal) of section. See section **1.4.1.2 Physical and Virtual Addresses**.
❹ Size in bytes (hexadecimal) of section.

❺ File offset (hexadecimal) to section data. Offset is zero if no data for the section.
❻ Section header flags. See section **1.4.4 Section Headers**, Table 1.7.
❼ Number of line number entries associated with the section. See section **1.4.6 Line Number Information**.
❽ File offset (hexadecimal) to the start of the line number entries for the section.
❾ Number of relocation entries in the section. See section **1.4.5 Relocation Information**.
❿ File offset (hexadecimal) to the start of the relocation entries for the section.

5.2. Object File Size Utility

Name

size68 – Sierra Systems Object File Size Utility

Syntax

size68 `[-ltu] [`*`obj_files`* `. . . ]`

Description

The **size68** utility takes Sierra Systems object files as arguments and prints size and address information on each file and its sections. The object files *obj_files* specified on the command line can be either relocatable or absolute COFF object files. If no file name is specified, the file **x.out** is assumed. If a file name without an extension is specified, the extension **.out** is assumed if the file cannot first be found without the extension. If more than one file is specified, **size68** also prints the total sizes of all files and all sections with like names.

Command line flags are:

`-l` Print **LOAD** address for each section. The default is to print only the virtual address of each section.

`-t` Suppress printing of section totals.

`-u` Print usage information.

Example

The following command displays the size and address information for each section in the executable file **hello.out**:

```
size68 hello.out
```

The output from the above invocation of **size68** is as follows:

```
Section     Address    Length    Decimal
-------     -------    ------    -------
.text          5000      17dc       6108
.ld_tbl        67dc        48         72
.data          6824         8          8
.bss           682c        e0        224
Total:                   190c       6412
```

Index

E

F

H

I