AT&T

# DSP16XX
# C Language Compiler
# User Manual

**Beta Release**

**February 1994**

# FOREWORD

This manual contains detailed information on the use and application of the DSP16XX C Language Compiler, which provides high-level language support and a software development environment for the DSP1600 family of devices. Specifically, this manual applies to the beta release of the compiler. Although the manual contains numerous examples, it is not intended for use as a C language tutorial. For more information on the C language and programming concepts, refer to a standard ANSI C reference book such as Kernighan and Ritchie's *The C Programming Language* (second edition, New York: Prentice-Hall).

Additional information on the digital signal processor product line is available in the form of manuals, data sheets, and application notes.

# DSP16XX C LANGUAGE COMPILER

# CONTENTS

## CHAPTER 1. INTRODUCTION



## CHAPTER 2. USING THE C COMPILER

## CHAPTER 3. CC1600 C LANGUAGE FEATURES



## CHAPTER 4. PROGRAM DEBUGGING



## CHAPTER 5. LIBRARIES



## CHAPTER 6. PROGRAMMING HINTS

## CHAPTER 7. RUNTIME ENVIRONMENT



## CHAPTER 8. INTERFACING C WITH ASSEMBLY LANGUAGE

## CHAPTER 9. HARDWARE/SOFTWARE INTEGRATION



## APPENDIX A. LIBRARIES REFERENCE



## APPENDIX B. RUNTIME EMULATION LIBRARY REFERENCE



## APPENDIX C. OPTIMIZATIONS



## APPENDIX D. AT&T DSP16XX C COMPILER COMMAND LINE REFERENCE

## APPENDIX E. C LANGUAGE EXTENSIONS



## APPENDIX F. COMMON ERRORS AND WARNINGS



## INDEX

# CHAPTER 1. INTRODUCTION

## CONTENTS

# 1. Introduction

The AT&T DSP16XX C Language Compiler translates ANSI-standard C language files into efficient DSP16XX assembly-language files. The DSP16XX compiler has been validated for conformance to the ANSI C specification, using the Plum-Hall test suite. The compiler supports the AT&T DSP1600 family of digital signal processors (DSPs).

The DSP16XX compiler is an optimizing C compiler based on GCC (GNU C Compiler). It is designed to compile and produce assembly code from C code; the resulting code compares favorably to code produced by programming in assembly language. In addition, the compiler provides a powerful facility for interfacing assembly code with C code so that real-time applications can be written in assembly language. The compiler also performs an extensive set of global optimizations, including loop optimizations.

Several libraries of commonly used arithmetic and signal processing functions are included with the C compiler. The C compiler allows application developers to generate efficient code without extensive knowledge of the DSP16XX architecture and instruction set.

No previous experience with digital signal processors is needed in order to use this document, but an understanding of digital signal processing concepts and assembly language is recommended, as well as some knowledge of C language programming.

## 1.1 Conventions Used in This Manual

This manual uses the following conventions:

| | |
|---|---|
| `Courier type` | Courier type denotes a command, a code example, or a response from the system. |
| *`Courier italic`* | Courier italic type denotes a file name on a command line, or variables or options that exist for a command. |
| **Bold type** | Bold type indicates the name of a command, an option, a routine, or a keyword. In text, bold type also represents a file name or a register name that is being described or referred to, or it is simply used for emphasis. |
| *Italic type* | Text shown in italic type stands for something that the user enters, such as a variable or a file name. |
| Brackets [ ] | Brackets indicate something optional that the user types in the location in which the brackets appear. For example, the command `cc1600 [options]` indicates that the user may add one or more compiler options where `[options]` appears. |
| Registers | A 32-bit register (for example, **a0**) contains two independent 16-bit halves, which are referred to as high and low halves (for example, **a0** contains **a0h**, the high half, and **P**, the low half). |
| DSP16XX | References to the DSP16XX apply to all members of the DSP1600 family of processors: the DSP1610, the DSP1616, etc. |

## 1.2 Overview of the DSP16XX Architecture

The DSP16XX Digital Signal Processor architecture is made up of the DSP1600 Core Processor, a dual-port RAM, ROM, and several peripheral blocks. The core contains the data arithmetic unit (DAU), the memory addressing units, the cache, and the control section. The core is a building block for designing new digital signal processors.

The DAU is the main computational execution unit of the processor. It can perform a 16 x 16 multiply, a 36-bit ALU operation, and two 16-bit data fetches from memory in a single instruction cycle. The DAU is made up of two input data registers, the multiplier, two accumulators, the ALU, and various control registers. Either of the two 36-bit accumulators can accumulate the product from the multiplier. The data in these accumulators can be directly loaded from memory (or stored to it) in 16-bit words. The ALU supports a full set of arithmetic and logic operations on either 16- or 32-bit data. Since a standard set of ALU conditions can be tested for conditional branches and subroutine calls, the processor functions as a powerful 16- or 32-bit microprocessor for logical and control applications. An optional bit manipulation unit (BMU) is provided on some DSP16XX devices for accelerating signal coding algorithms. It performs full 36-bit barrel shifting, normalization, and bit field extraction or insertion of data in the accumulators. Two alternate accumulators provide storage for 36-bit data.

Two addressing units support high-speed, register-indirect memory addressing with postmodification of the register. Four address pointer registers can be used for either read or write addresses to the RAM without restrictions. One address register is dedicated to the instruction/coefficient memory space for table lookup. Direct data addressing is supported for 16 key registers. A unique compound addressing mode swaps data between a register and memory with only two instruction cycles. Immediate addressing can be done with a 9-bit address in a one-cycle instruction, or with a 16-bit address in a two-cycle instruction.

For more information about the DSP16XX architecture, refer to the *DSP16XX Digital Signal Processor Information Manual*

## 1.3 Installing the C Compiler

The *DSP16XX C Compiler Release Notes* contain the information needed to install and set up the C compiler on various computer systems.

## 1.4 The cc1600 Compilation System

This section describes the cc1600 compilation system and explains the functions of each of the components of the system. This section also lists and briefly describes the compilation tools provided with the C compiler.

### 1.4.1 The Compilation Process

Figure 1-1 shows the cc1600 compilation system.

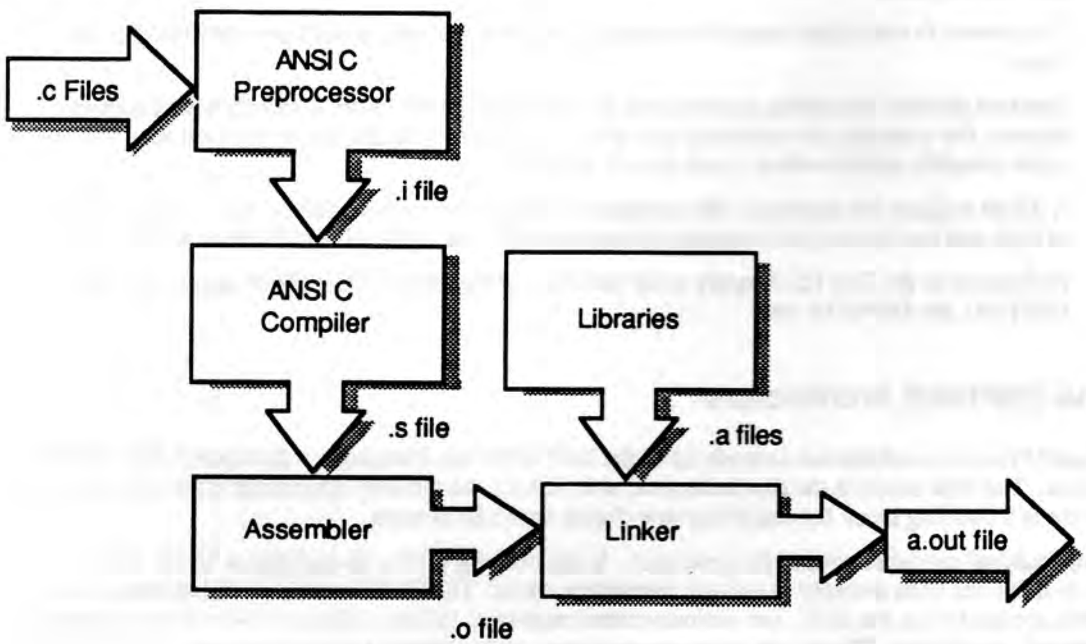



**Figure 1-1. Compilation Process**

### 1.4.2 Compilation Tools

The following compilation tools are used to develop signal processing applications:

- the **cc1600** compiler
- the **cpp1600** preprocessor
- the **as1600** assembler
- the **ar1600** archiver
- the **ld1600** linker
- the **win16XX** windowing simulator
- the **cmd16XX** command line simulator

The **cc1600** compilation driver tool is the user interface to the compilation process. It directs each input file (whether C, assembly, or object) to the appropriate compilation tool. **cc1600** is a single command that orchestrates the compiling, assembling, and linking of programs. **cc1600** functions like the *UNIX* * C compiler **cc** in that it passes input files to the appropriate phase of the compilation process based on the type of the specified file. For example, C source files (with the extension **.c**) enter the compilation process at the C preprocessor, whereas object files (with the extension **.o**) enter the compilation process at the linker level.

**cpp1600** is the ANSI C preprocessor. It allows the use of include files and conditionally compiled code with the C compiler.

**as1600**, **cmd16XX**, **ld1600**, and **win16XX** are referred to as the support software tools. These tools are described more extensively in the *DSP16XX Support Tools Manual.*

**as1600** is the stand-alone assembler for the DSP16XX. It uses a DSP16XX assembly-language file as input and creates a relocatable object file.

**ar1600** is the library archiver for the DSP16XX. It takes relocatable object files and creates library archive files to be input to the linker.

**ld1600** is the linker. It receives the output files from the assembler and archiver and combines them into a single executable file. The memory maps of the DSP16XX are also specified at this stage, as directed in the ifiles (linker information files) that are input to the linker.

**win16XX** is the windowing simulator. It receives the executable output file from the linker. For a complete list of the capabilities of the windowing simulator, refer to the *DSP16XX Support Tools Manual.*

**cmd16XX** is the command line simulator. It receives the executable output file from the linker. For more information on the command line simulator, refer to the *DSP16XX Support Tools Manual.*

### cc1600 Front End

Figure 1-2 illustrates the operations that take place in the cc1600 front end.

* UNIX is a registered trademark of UNIX Systems Laboratories, Inc.

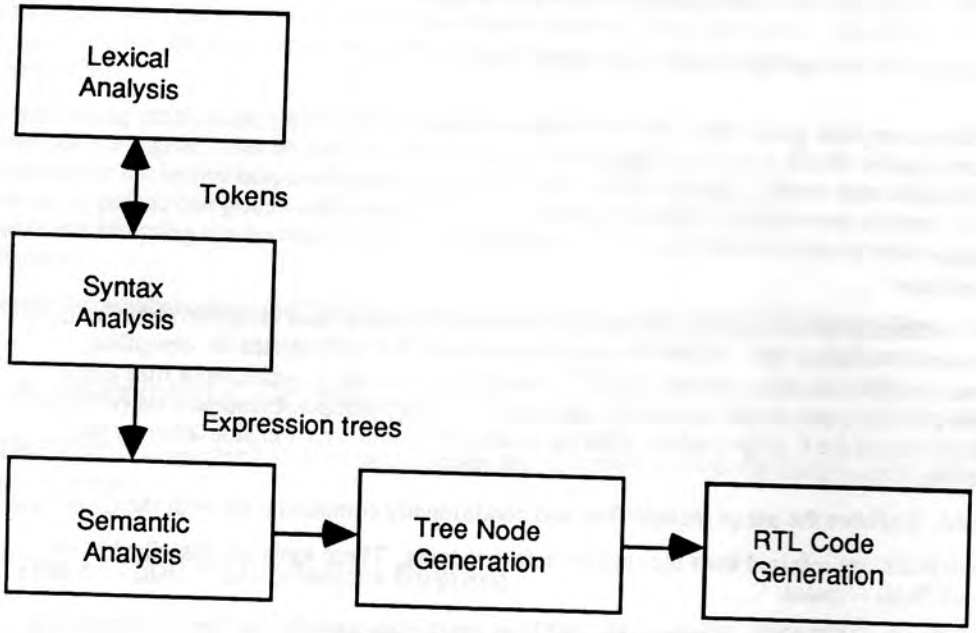



**Figure 1-2. Structure of the cc1600 Front End**

The cc1600 front end performs five discrete operations on the code:

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Tree node generation
5. RTL (register transfer language) code generation

In **lexical analysis**, cc1600 separates the input file into pieces, or tokens. Each class of tokens is then given a unique internal representation in the analyzer. Finally, the tokens are supplied to the syntax analyzer.

The **syntax analyzer** determines the overall structure of the source program. The analyzer groups the tokens into larger classes according to their syntax (**semantic analysis**). Once these classes are created, the analyzer creates syntax trees (**tree node generation**).

For example, if the input file says **c = a+b**, the syntax analyzer groups the tokens for this statement according to their relationships, as shown in Figure 1-3.

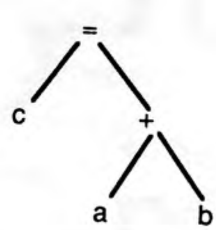



**Figure 1-3. Sample Syntax Tree**

The information in the syntax tree is nearly at the source code level. The structure of the tree can represent a variable, a function, an array, etc., and each tree describes a single expression. There are a number of different kinds of tree nodes, and each kind describes a certain kind of relationship (for example, a constant, an expression, or a global declaration).

Next, cc1600 creates RTL code from the syntax trees. RTL (Register Transfer Language) is an intermediate language used by the cc1600 back end. Once the RTL code has been generated and the semantic analysis of the program is complete, the RTL code is transferred to the cc1600 back end.

### cc1600 Back End

The cc1600 back end performs a number of operations on the RTL code:

- Optimization assumptions
- Jump optimization
- Common subexpression elimination
- Loop optimization
- Data flow analysis
- Instruction combination
- Register class preferencing
- Local register allocation
- Global register allocation
- Reloading
- Final output processing

Some of these operations occur more than once during the course of the back-end processing.

When the cc1600 back end receives the RTL code from the front end, the back end first subjects the code to optimization assumptions. This process simplifies some of the RTL code structures and performs some optimizations on the RTL code. The code then passes on to jump optimization.

Jump optimization analyzes the structure of the program and simplifies any jump instructions, such as jumps to the next instruction in the code, or jumps to other jumps. This process also deletes code that cannot be reached using the instructions in the code, as well as labels that are declared but never referred to, or move instructions that have no function.

Next, the code undergoes common subexpression elimination (CSE). This process examines the code for subexpressions that may be used more than once, and combines these subexpressions. This helps speed up the resulting assembly code by making sure the program needs to compute a subexpression only once.

Loop optimization examines loops in the code. If a constant is declared inside the loop, the step moves the constant out of the loop. This phase can also "unroll" the loops into straight RTL code, thus eliminating costly compare instructions.

Data flow analysis divides the program into blocks and examines each block. It deletes computations whose results are never used in the program and creates addresses for autoincrement and autodecrement operations.

Instruction combination examines the data flow of the instructions in the program and combines related instructions into single instructions.

Register class preferencing scans the RTL code and assigns each register class to a pseudo register. Pseudo registers are temporary (virtual) registers, as opposed to actual physical memory locations. Local register allocation then allocates physical registers to pseudo registers that are used within only one basic block of the program.

After the local registers have been allocated, global register allocation allocates physical registers to all the remaining pseudo registers. These pseudo registers are generally used for more than one block of the program.

The reloading step assigns stack slots to any pseudo registers that did not receive physical registers in the allocation steps. (This step does not necessarily result in spill code.) This step also scans the code for instructions that are invalid because a value has failed to end up in a register or has ended up in the wrong register class. The reloading process generates code to copy any such values into the correct locations. Spill code is created at this stage, if necessary. The reloading phase can also eliminate the frame pointer and replace it with references to the stack pointer.

In the final phase, the cc1600 back end generates assembler code from the RTL code for each function. This step deletes unneeded test and comparison operations, performs machine-specific optimizations on the code, and generates entry and exi sequences for each function.

## 1.5 Applicable Documentation

The DSP1600 family documentation set provides specific information on various members of the DSP1600 product family. Contact your AT&T Account Manager for the latest issue of any of the following documents. These documents comprise th documentation set:

The *DSP16XX Digital Signal Processor Information Manual* is a reference guide for the DSP16XX. It describes the architecture, instruction set, and interfacing requirements for the processor.

The *DSP16XX C Language Compiler Release Notes* explains how to install the C compiler on your system. The release notes also provide additional release-dependent information.

The *DSP16XX Digital Signal Processor Data Sheet* provides up-to-date timing requirements and specifications, electrical characteristics, and a summary of the instruction set and device architecture.

The *DSP16XX Support Tools Manual* provides the information necessary to install and use the DSP16XX support software. This manual is also useful when working with the hardware development systems, since the support software provides an interface between the host computer and the development system.

## 1.6 Manual Summary

Chapter 2, Using the C Compiler, explains how to invoke and use the C compiler, how to set up the environment variables, and how to use some common options.

Chapter 3, cc1600 C Language Features and Extensions, describes the C language features and extensions available for the compiler. It also describes the various data types used with the compiler.

Chapter 4, Program Debugging, describes how to use the C compiler with the simulator. This chapter also contains a number of general hints and techniques for programming and debugging.

Chapter 5, Libraries, explains how to use the libraries provided with the compiler, as well as how to create libraries.

Chapter 6, Programming Hints, contains information on programming techniques.

Chapter 7, Runtime Environment, describes the various runtime memory models for the C compiler. This chapter also contains information on register usage and function calling conventions.

Chapter 8, Interfacing C with Assembly Language, describes how to combine assembly language and C instructions in programs.

Chapter 9, Hardware/Software Integration, describes how to use linker information files. This chapter also provides detailed information on startup files and using initialized data.

Appendix A, Libraries Reference, consists of detailed reference material on the provided libraries.

Appendix B, Runtime Emulation Library Reference, lists the functions included in the runtime emulation library.

Appendix C, C Compiler Optimizations, contains a list of optimizations for the C compiler.

Appendix D, AT&T DSP16XX C Compiler Command Line Reference, contains the command line reference for the C compiler. It lists all of the available compiler options.

Appendix E, C Language Extensions, lists the extensions to ANSI C that are provided by the compiler.

Appendix F, Common Errors and Warnings, is a list of common error codes generated by the C compiler.

# CHAPTER 3. cc1600 C LANGUAGE FEATURES

## CONTENTS

# 3. cc1600 C Language Features

This chapter describes the data types that the C compiler supports for the DSP16XX. It also discusses some of the features provided by ANSI C.

## 3.1 Data Types

The cc1600 compiler supports both integer and floating-point data types. This includes the following integer data types:

- char
- unsigned char
- short
- unsigned short
- int
- unsigned int
- long
- unsigned long

The compiler uses 16- and 32-bit two's complement numbers to represent the C integer types. Long-word values are stored in big-endian format in memory.

The compiler supports the following floating-point data types:

- float
- double

Both the float and double data types are 32-bit IEEE single-precision numbers, as specified by the IEEE standard for binary floating-point arithmetic, IEEE 754-1985. As with the integer long-word values, float and double values are stored in big-endian format.

## 3.2 C Language Data Type Sizes

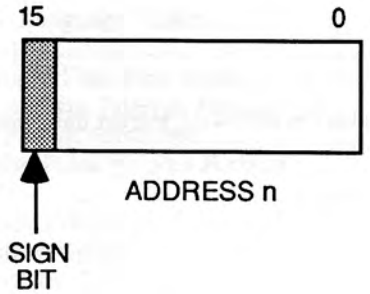
Table 3-1 shows the physical size of the various data types for the DSP16XX. Because the DSP16XX is a 16-bit word machine, the minimum data size is 16 bits.

### Table 3-1. Data Type Sizes

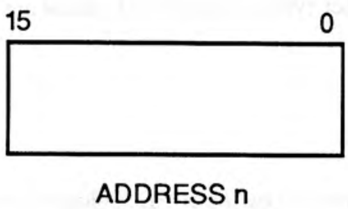
| Data Type | Size |
|---|---|
| char | 16 bits |
| short | 16 bits |
| int | 16 bits |
| long | 32 bits |
| float | 32 bits |
| double | 32 bits |
| pointers | 16 bits |

## 3.3 Layout of Data Types in Memory

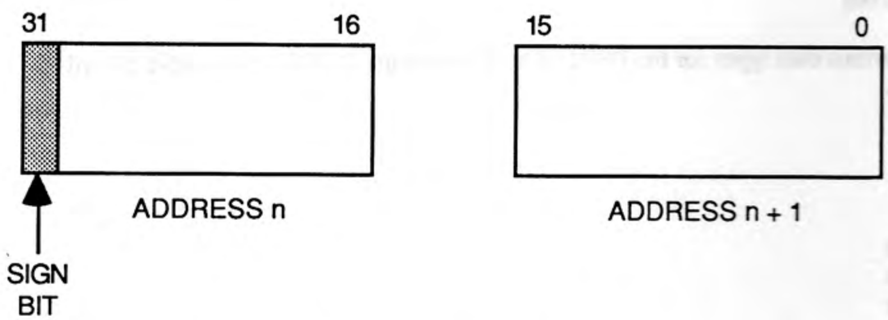
The **char**, **short**, **int**, and **pointers** data types have the following layout:



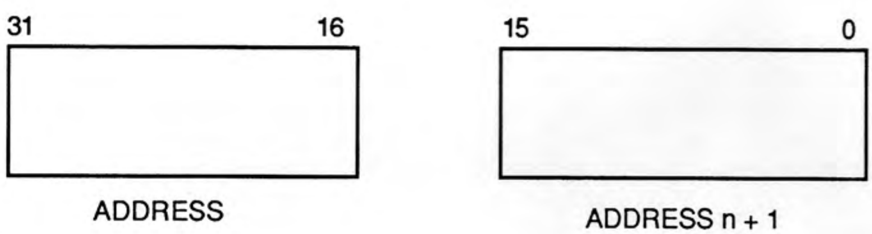

The **pointer**, **unsigned char**, **unsigned short**, and **unsigned int** data types have the following layout:



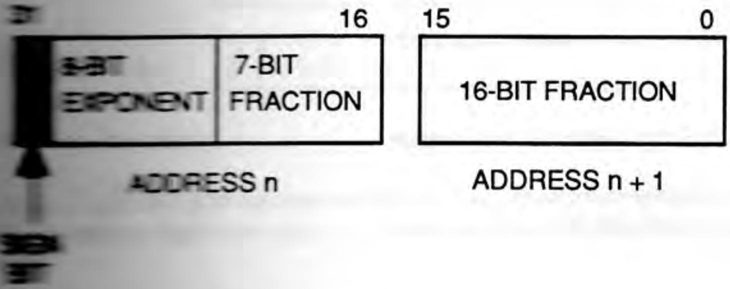

The **long** data type has the following layout:




The **unsigned long** data type has the following layout:



The float and double data types have the following layout:




## 3.3 Floating-Point Arithmetic

As previously mentioned, the float and double data types for the DSP16XX are based on the IEEE standard for binary floating-point arithmetic, IEEE 754-1985. This section summarizes the standard as it affects these data types.

The IEEE standard specifies the following for its floating-point system:

- The format for floating-point operands
- The accuracy of arithmetic results for addition, subtraction, multiplication, division, square roots, obtaining remainders, and comparison operations
- Conversion between integers and floating-point numbers
- Conversion between different floating-point formats
- Conversion between binary floating-point and decimal numbers
- Floating-point exceptions

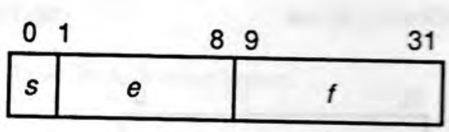
### 3.3.1 Floating-Point Number Formats

There are two basic formats for radix 2 (binary) floating-point operands: 32 bits and 64 bits. The DSP16XX C compiler supports only 32-bit (single precision) floating-point numbers.

Since the 24-bit mantissa for normalized numbers $m$ is limited to the range $0<m\leq1$, the bit to the immediate left of the binary point is always 1, and is therefore not represented in the floating-point word (this bit is referred to as the implicit bit). The remaining 23 bits of the fractional part $f$ of the mantissa are represented in the floating-point word, as illustrated in Figure 3-1.

The 8-bit exponent $e$ lies in the range $0\leq e\leq255$. The value of $e=0$ is reserved for the number 0.0 and for denormalized numbers (described below). The value of $e=255$ is reserved for $\infty$ and for NaN (Not a Number).The remaining values of the exponent are used to represent normalized numbers. A bias of -127 is applied to the exponents of normalized numbers, resulting in the range (-126, +127). Since this range is unbalanced, the reciprocal of a floating-point number with an exponent of +127 underflows the range of normalized numbers. Therefore, denormalized numbers are supported so that the reciprocal of all normalized floating-point numbers can be represented without causing exponent underflow.

Figure 3-1 illustrates the single-precision format as defined by the standard.




SINGLE-PRECISION FORMAT

**Figure 3-1 IEEE Single-Precision Format**

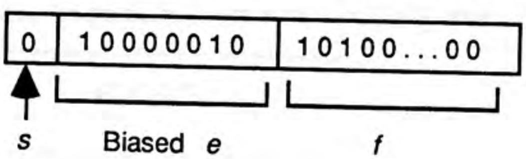
The single-precision format is a 32-bit format for binary floating-point numbers. The format consists of a 1-bit sign $s$, an 8-bit biased exponent $e$, and a 23-bit fraction $f$. The value $V$ of the number is as follows:

1. $V$ = NaN (not a number), if $e = 255$ and $f \neq 0$.
2. $V = -1^s \times \infty$, if $e = 255$ and $f = 0$; that is, $V = \pm\infty$.
3. $V = -1^s \times 2^{e-127} \times 1.f$, if $0 < e < 255$.
4. $V = -1^s \times 2^{-126} \times 0.f$, if $e = 0$ and $f \neq 0$ (denormalized numbers).
5. $V = -1^s \times 0$, if $e = 0$ and $f = 0$; that is, $V = \pm 0$.

The following example illustrates case 3.

$+13 = +2^3 \times 1\,.\,1\,0\,1\,0\,0 \ldots 0$ unbiased exponent

After adding the bias of 127 to the unbiased exponent (3), the single-precision format of the number is as follows:



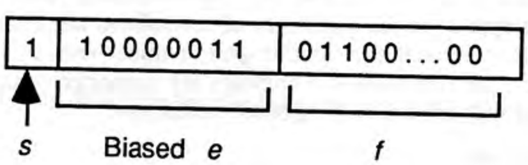

Therefore, using case 3,

$$V = -1^0 \times 2^{130-127} \times 1\,.\,1\,0\,1\,0\,0 \ldots 0$$
$$= 1 \times 2^3 \times 1\,.\,1\,0\,1\,0\,0 \ldots 0$$
$$= 13$$

A second example using case 3 is:

$-22 = -2^4 \times 1\,.\,0\,1\,1\,0\,0 \ldots 0$

After adding the bias, the single-precision format of the number is:




Therefore, using case 3:

$$V = -1^1 \times 2^{131-127} \times 1\,.\,0\,1\,1\,0\,0 \ldots 0$$
$$= -1 \times 2^4 \times 1\,.\,0\,1\,1\,0\,0 \ldots 0$$
$$= -22$$

The following example shows addition in single-precision format. The operation is (+1) + (+1).

| [illegible] | | | |
|---|---|---|---|
| [illegible] = | 0 | 01111111 | 000...00 |

| [illegible] | | | |
|---|---|---|---|
| [illegible] = | 0 | 01111111 | 000...00 |

[illegible] the sum is

[illegible] ...00

[illegible] significant bit is a 1 for each number. Postnormalization is required for this operation, and this yields the [illegible]

| [illegible] | | | |
|---|---|---|---|
| [illegible] = | 0 | 10000000 | 000...00 |

## [illegible] Language Features and Restrictions

[illegible] some of the features of ANSI C that differ from traditional C language, and the programming [illegible] by the use of ANSI C.

### [illegible] Prototypes

[illegible] feature that the ANSI C committee added to the C language is function prototypes. This feature [illegible] of the return type of a function, but also the number and types of parameters. This promotes [illegible]

[illegible]

```
[illegible] arg1, long arg2);
[illegible]

[illegible] arg1, long arg2)
[illegible]
    [illegible]+arg2);
[illegible]

[illegible]
[illegible]
    [illegible] = %d\n", add(20,30));
[illegible]
```

[illegible] this example. The first, **add**, takes two long arguments and produces a long result. The second [illegible] (as signified by **void**) and returns nothing. In the example, the function **foo** is attempting to pass [illegible] **add**. The compiler will automatically promote the arguments to type long.

[illegible] would not occur; this could result in a programming error. With a prototype, if an [illegible] be converted using the ANSI C promotion rules, the compiler generates a warning. If not [illegible] too many arguments are passed to the function **add**, the compiler generates an error. [illegible] present, the compiler could not perform this checking.

### [illegible] Keyword

[illegible] may be added to the declaration of a variable to make that variable read-only (as opposed to read/write). [illegible]

```
const int year = 1990;
const int table[] = {1, 2, 3, 4};
```

Because it cannot be assigned, a constant must be initialized. Declaring something as a constant ensures that its value will not change during the scope of the function. For example:

```
year = 1991;        /*error*/
year++;             /*error*/
```

In this case, the compiler would generate a diagnostic error.

### 3.5.3 The `volatile` Keyword

The **volatile** keyword was added to the language to indicate to the compiler not to attempt optimization for the indicated variable. The **volatile** keyword was added mainly for embedded systems where an I/O location (for example) must be read repeatedly. In this case, using the keyword would prevent the compiler from reading the value once, then placing the value in a register so that the compiler can perform the subsequent read operations on the register. When the compiler sees the **volatile** keyword, it cannot perform any optimizations on the variable. For example:

```
volatile int serial_line;

main()
{
   int*p;

   p = &serial_line;
   while (*p != 0)
      process (p);
}
```

In this example, the compiler must honor the **volatile** keyword and read from the memory location each time through the loop.

## 3.6 GNU C Extensions

The DSP16XX C compiler is based on GNU C, and thus makes use of the features and extensions of GNU C. For a complete listing and description of these extensions, see Appendix E, C Language Extensions.