COSMIC, sarl

Version 3.32

January 1989

C LANGUAGE SPECIFICATION

for Microcontroller Environments

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

CHAPTER 2

THE C LANGUAGE

CHAPTER 3

USAGE CONSIDERATIONS

PREFACE

The **C Language Specification for Microcontroller Environments** describes the C language as implemented by Whitesmiths, Ltd. for microcontrollers. It is intended as a reference tool for programmers using Whitesmiths C to write applications programs targeting microcontrollers. This manual assumes an understanding of programming fundamentals and a working knowledge of C.

For an explanation of how to use Whitesmiths' C compiler in your specific programming environment, refer to your **Cross Compiler User's Guide.**

Organization of this Manual

This manual is divided into three chapters.

Chapter 1, "Introduction," describes the basic organization of the C compiler and the notational conventions used in this manual.

Chapter 2, "The C Language," describes the details of Whitesmiths' current implementation of C.

Chapter 3, "Special Considerations," discusses issues that relate to good programming style and language usage.

This manual also contains an **Index.**

CHAPTER 1

Introduction

This chapter describes the basic organization of the C cross compiler, the ANSI C language standard to which Whitesmiths C conforms closely, and the notational conventions used in this manual.

Compiler Organization

Whitesmiths' C compiler is a set of three programs that take as input, to the first of the programs, one or more files of C source code, and produce as output, from the last of the programs, assembly language code that will perform the semantic intent of the source code. Output from the files may be separately compiled, and then combined at load time to form an executable program, or C subroutines can be compiled for later inclusion with other programs. One can also view the compiler as a vehicle for implementing an instance of an abstract C machine, i.e., a machine that executes statements in the language defined by some standard.

Language Standard

For many years, the C language standard has been generally accepted to be "Appendix A" of Kernighan and Ritchie, The C Programming Language, Prentice-Hall 1978. More recently, the ANSI X3J11 Language Subcommittee has published the C Information Bulletin (April 30, 1985), which describes the preliminary draft proposed standard for C. This version of Whitesmiths' C compiler has been significantly enhanced to bring it into closer conformance with the emerging ANSI standard. This version of Whitesmiths C also offers a number of useful extensions to C, nearly all of which can be turned on or off by various compile time options. Notes that appear in square brackets are used to warn of dramatic alterations to the language that occur when these options are invoked.

Major Features of Whitesmiths C

The ANSI Draft Proposed Standard for C is upward compatible with the original definition of the C language as specified in Appendix A of the book **The C Programming Language** by B.W. Kernighan and D.M. Ritchie. However, Standard C provides additional reliability features and aids for the embedded systems programmer. The most important new features of Standard C are described below.

Function Prototyping

You can now declare the types of arguments to a function, as well as return types, in a declaration called a function prototype. Function arguments are then type checked and converted properly on actual calls. This helps eliminate the largest source of errors in multi-file C programs.

Type Modifiers

Standard C provides two "type modifiers," **const** and **volatile,** which give the compiler more information about how your application accesses data objects. You use the **const** type modifier to inform the compiler that a data object is not to be modified. You use the **volatile** type modifier to declare memory mapped input/output control registers and data objects accessed by signal handlers. The compiler will not optimize references to volatile data objects.

Your Cross Compiler User's Guide provides tips on how to use the type modifiers in programs targeting your specific microcontroller environment.

Compile Time Type Checking

Whitesmiths C provides a compile time option that allows you to specify one of four levels of type checking. You can choose the level appropriate to each application.

- The strictest level enforces "lint style" type checking that is stronger than Standard C requires. This level of type checking helps remove obscure, machine-dependent or otherwise peculiar type-mismatch

programming errors.

- The next level checks for rigorous conformance with the ANSI/ISO standard.
- The next level tests for conformance with Standard C, but permits Whitesmiths extensions.
- The last level is for compiling older style C programs, which require less strict type checking.

C Runtime Support

Whitesmiths C for microcontroller environments provides a subset of the ANSI Standard C library that is appropriate for embedded applications. Each library function is provided in source code form in the binary package. Support is provided for:

- character handling
- mathematical functions
- nonlocal jumps
- input/output
- string handling
- memory management

In addition, Whitesmiths provides an integer only library as well as a floating point library. You can therefore select smaller integer only functions if your application does not require floating point support. Your Cross Compiler User's Guide describes the runtime support for your specific environment.

Other Enhancements

Whitesmiths C also provides:

- enumeration types
- void types
- structure assignment
- structures as function arguments

- functions returning structures

Whitesmiths Extensions to ANSI C

Whitesmiths has implemented numerous useful extensions to the C language, all of which can be easily enabled or disabled. These extensions include:

- inline generation of single byte instructions;
- inline generation of assembler code, by using a special _asm() function;
- definition of functions that save volatile registers for handling exceptions and interrupts (see your Cross Compiler User's Guide for implementation-specific information on this topic);
- definition of functions and data objects at absolute memory addresses, to prelocate interrupt handlers and define memory mapped input/output in C (see your Cross Compiler User's Guide for implementation specific information on this topic);
- ability to specify a range of values in a single **case** label;
- the special preprocessor macro **$if** lets you conditionally include text in a macro expansion;
- you can precede any scalar data initializer with a repeat count enclosed in square brackets **'[,]'**;
- substitution of macro arguments within string constants;
- optional specification of the UNIX "common model," which permits external data objects with no defining instance;
- various bitfield manipulation operations. Bitfields can be character-based or integer-based. It is possible to mix both kinds of bitfields in the same program. Bits in a bitfield may be numbered from least to most significant, or in the reverse order, at the user's option;
- single precision floating point. The compiler can handle all floating point numbers in single-precision format, rather than coercing them to doubles in floating-point mathematics. Calculations are made in

single-precision and arguments are passed as floats, with corresponding decrease in storage requirements and (some) speedup in mathematical processing.

Notational Conventions

This section describes the notational conventions that appear throughout this manual.

Boldfaced Print

Boldface print is used to specify the following:

1) utility names
2) function names
3) flags to utilities
4) commands used with interactive utilities
5) C keywords or reserved words
6) formulas
7) variables and optional or user-defined command line elements described in "metanotion" form (i.e. enclosed in angle brackets **< >**, square brackets **[ ]**, or both). The meanings of these notational conventions are discussed under "Notational Shorthand" below.
8) example filenames and values for variables, etc. that are referred to in explanations
9) specific values or characters used in commands or code, or referred to in explanations.
10) anything that the user types

Specifying Ranges

[ ] Square brackets are used to specify the range of a particular element (referred to as a "closed" interval). For example, the range specification for the closed interval **[2, 6]** means that the element can be any number between 2 and 6, including both 2 and 6.

[ ) The parenthesis in this notation (referred to as a "half-open interval") indicates that whatever value appears on the right-hand side of the set (or the left-hand side if that's where it appears) is not part of the valid set of values. For example, **[3, 7)** means that the valid range of the element in question can be any number between 3 and 7, including 3 but not 7. Alternatively, valid numbers in the range **(1, 5]** would be 2, 3, 4, and 5.

( ) Parenthesis are used to specify the range of a particular element (referred to as an "open" interval). Given the open interval **(0, 5)**, therefore, the valid range of numbers is 1, 2, 3, and 4.

Capital Letters

CAPITAL LETTERS are used for names which must be entered in all capital letters, or that appear as such in source code, header files, and so forth. Capitalization is not used simply for extra emphasis.

Notational Shorthand

Grammar, the rules by which syntactic elements are put together, is important to the remaining discussions. Some simple shorthand is used throughout this manual to describe grammatical constructs.

A name enclosed in angle brackets, such as **<statement>**, is a "metanotion", i.e., some grammatical element defined elsewhere. Any sequence of tokens that meets the grammatical rules for that metanotion can be used in its place, subject to any semantic limitations explicitly stated. Just about any other symbol stands for itself, i.e., it must appear literally, like the semicolon in

<statement> ; <statement>

Exceptions are the punctuation **[**, **]**, **]***, and **|**; these have special meanings unless made literal by being enclosed in single quotes.

Brackets surround an element that may occur zero or one time. The optional occurrence of a label, for instance, is specified by:

[<label> **:** **]** <statement>

This means that metanotion **<label>** may (but need not) ap-

pear before **<statement>**, and, if it does, it must be followed by a literal colon. To specify the optional, arbitrary repetition of an element, the notation **[ ]*** is used. A comma-separated list of **<ident>** metanotions, for example (i.e., one instance of **<ident>** followed by zero or more repetitions), would be represented by:

<ident> [, <ident>]*

Vertical bars are used to separate the elements in a list of alternatives, exactly one of which must be selected. The line:

char | int | long | short

requires the specification of any one of the four keywords listed.

CHAPTER 2

THE C LANGUAGE

This chapter describes in detail Whitesmiths' implementation of the C Language. It discusses keywords, punctuation, naming objects, declaring identifiers, specifying initial values, writing C statements and expressions, and the preprocessor.

Syntax

At the lowest level, a C program is represented as a text file, consisting of lines each terminated by a newline character. Any characters between **/*** (not in a character or string constant) and ***/** inclusive, including newlines, form a comment, which is replaced by a single blank character. A newline preceded by **** is discarded, so that long tokens, such as string constants, may be continued on multiple lines. The compiler cannot deal with a text line longer than 511 characters, either before or after the processing of comments and continuations.

Text lines are broken into tokens, or strings of characters possibly separated by whitespace. Whitespace consists of one or more of the motion control characters **space, carriage return, form feed, tab,** or **newline.** Its sole effect is to delimit tokens that might otherwise be merged. Tokens take several forms, each of which is described below.

Identifiers

An identifier consists of a sequence of letters, dollar signs, underscores, or digits, beginning with a non-digit. [**Note:** Use of the dollar sign in identifiers is an extension, which may be disabled.] Uppercase letters are distinct from lowercase letters, and no more than 127 characters are significant when comparing identifiers. More severe restrictions may be placed on external identifiers by the world outside the compiler (see the section "Differences Among C Compilers" in Chapter 3 for details). There are also a number of identifiers reserved for use as

keywords. These are:

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

Examples of legal identifiers are:

abc
Terpsichorean
new_one
a10b20

[**Note:** The identifier **_asm** is reserved as a function name. The function _asm() is used to generate in-line assembly code in a C program.]

[**Note:** **$noside** is an extension which may be disabled.]

Integer Constants

An integer constant consists of a decimal digit, followed by zero or more letters and digits. If the leading characters are **0x** or **0X**, the constant is hexadecimal and may contain the letters **a** through **f**, in either case, to represent the digit values 10 through 15, respectively. Otherwise a leading **0** implies octal, which may not contain the digits **8** or **9**. A non-zero leading digit implies decimal. Any of these forms may end in **l** or **L** to specify a **long** constant, as well as **u** or **U** to specify an **unsigned** constant. At most one of each of these qualifiers may occur, in any order, in a constant. A constant is also made **long** if

a) a decimal constant cannot be properly represented as a signed integer, or

b) any other constant cannot be properly represented as an unsigned integer. A constant is also made **unsigned** if

a) a non-decimal constant cannot be properly represented as a signed integer, and

b) it can be properly represented as an unsigned integer. Overflow is not diagnosed in either case. Examples of various ways of writing an integer constant:

integer	10, 012, 0xa
long	10L
unsigned long	10UL

Floating Constants

A floating constant consists of a decimal integer part, a decimal point, a decimal fraction part, and an exponent, where an exponent consists of an **e** or **E** and an optionally signed (+ or –) decimal power of ten by which the integer part plus fraction part must be multiplied. Either the decimal point or the exponent may be omitted, but not both; either the integer part or the fraction part may be omitted, but not both. Any of these forms may end in **l** or **L** to specify a **long double** constant, or in **f** or **F** to specify a **float** constant. If none of these forms appear, the constant is a **double**. Overflow is not diagnosed. Examples of float constants:

float	10.0F, 1e1F
double	10.0, 1e1
long double	1.0e+01L

Character Constants

A character constant consists of a single quote, followed by zero or more character literals, followed by a second single quote. A character literal consists of

a) any character except ****, **newline**, or **'**, the value being the target machine's representation of that character; a \ followed by up to three octal digits, the value being the octal number represented by these digits;

c) a \ followed by x or X, followed by up to three hexadecimal digits, the value being the hexadecimal number represented by the two low order digits; [**Note:** \X is an extension.]

d) a \ followed by one of the characters in the sequence **<a, b, t, v, f, n, r, (, !,), ^>** the value being the target machine's representation for the corresponding member of the sequence **<bell, backspace, horizontal tab, vertical tab, form feed, newline,** carriage

return, {, |, }, ~>; [Note: The characters in the set <(, !,), and ^> are an extension, which may be disabled.]

e) a \ followed by anything else; the value being the target machine's representation of that character. [**Note:** This last form is discouraged, since new "escape sequences" are defined in this fashion from time to time.] A **newline** is never permitted inside quotes (except when escaped with a \ for line continuation).

The value of the constant is an integer, base 256, the digits of which are the character literal values. A character constant consisting of a single character literal is effectively cast to type **char.** It may therefore be negative on some machines if the character constant is a non-printing character. A (multi-character) constant is **long** if it cannot be properly represented as an **int** or **unsigned int.** Overflow is not diagnosed. [**Note:** Character constants other than length one are often non-portable.] Examples of character constants are:

```
'a'     'A'     '3'
'\0'    '\n'    '\012'
```

String Constants

A string constant is just like a character constant, except that double quotes " are used to delimit the string. Thus, the sequence **\"** must be used to specify a double quote character literal, and a single quote need not be escaped. The value is the (secret) name of a NUL-terminated constant array of characters, the elements of which are initialized to the character literals in the string. String constants may have overlapping addresses. Examples of string constants are:

```
""
"Hello world!\n"
"\b indent"
"a\1&a\2&a\3"
```

Punctuation

Punctuation consists of predefined strings of one to three characters. The complete set of punctuation for C is:

```
!       )       .       =&      >)      \)
!=      *       /       =*      >=      \^
```

```
#       *=      /=      =+      >>      ^
%       +       :       =-      >>=     ^=
%=      ++      ;       =/      ?       {
&       +=      <       =<<     @       |
&&      ,       <<      ==      [       |)
&=      -       <<=     =>>     ]       |=
(       --      <=      =^      \!      ||
(<      -=      =       =|      \!!     }
(|      ->      =%      >       \(      ~
```

Punctuation in the sequence **<(< (| >) \! \!! \(\), \^ |)>** is entirely equivalent to the corresponding member of the sequence **<{ [} | || { } ~]>**. **[Note:** These mappings are an extension, as are the assigning operators with a leading =, to support non-ASCII terminals and early C programs. @ is an extension as well. All may be disabled.] The longest possible punctuation string is matched first, so that +++, for example, is recognized as "++, +" and never as "+, ++".

Other characters such as \ alone, are illegal outside of character or string constants, and are diagnosed as such.

Identifiers

Identifiers are used to give names to the functions, objects, and so forth created in a C program. Syntactically, an identifier is a sequence of letters, underscores, dollar signs, and digits, starting with a nondigit. For the sake of comparison, only the first 31 characters are significant. Externally published identifiers are often even more restricted (see the section "Differences Among C Compilers" in Chapter 3 for details).

There are several name spaces in a C program, so that the same identifier may have different meanings in the same extent of program text, depending on usage. Things such as **struct**, **union** and **enum** tags, members of **struct** or **union**, and labels will be considered separately later on. The bulk of this discussion concerns the name space inhabited by the names of type definitions, functions, and objects (which occupy storage at execution time).

Each such identifier acquires, from its usage in a C program, a precisely defined lexical scope, storage class, and type. The scope is the extent of program text over which the compiler knows that a given meaning holds for an identifier. The storage class determines both the lifetime of values assigned to an object and the extent of program text over which a given meaning holds for its identifier, whether the compiler knows it or not. The

type determines what operations can be performed on a function or object and how object values are encoded. These important attributes often interact.

Scope

There are two basic contexts in a C program -- inside a "program block" and outside it. A program block can be the entire body of a defined function, including its argument declarations, or any set of statements enclosed in braces { }.

If an identifier first appears outside a program block, its scope extends from its first appearance to the end of the file, except for contained program block in which that identifier is explicitly redeclared.

Outside of an explicit declaration, the only other place an identifier may legally first appear is inside a program block, within an expression, where the name of a function is required. In this case the identifier is implicitly declared to be **extern,** with type **function returning int.**

[**Note:** Functions and objects which have the storage class **extern** or **static** outside a program block are "remembered" for the sake of consistency checking, even in areas where they are properly out of scope. A program that depends on this behavior to avoid redeclaration is not portable.]

Storage Classes

There are several different storage class keywords, whose effect on declared identifiers may vary depending on context.

extern, when not previously declared, means that a given identifier names a function or object, the name of which should be published for use among any of the files composing the program that also publish the same name. That is, it has "program scope". The published name may be shortened to as few as six significant characters, and/or compressed to one case, depending on the target operating system; so whereas the compiler distinguishes between "Counted" and "counter," subsequent programs processing the compiled text may not. If previously declared, the earlier declaration must be either **extern** or a **static** outside a program block. The storage class is unaltered by the newer declaration, which refers to the same function or object. The lifetime of **extern** objects is the duration of program execution.

static, outside a program block, means that the name should not be published outside the file. That is, it has "file scope". If previously declared, the function or object must already have storage class **static.** Inside a program block, **static** means that the identifier names an object known only within the program block, less any local redeclarations. That is, it has "block scope". The lifetime of static objects is the duration of program execution, so the value of a local static is retained between invocations of the program block that knows about it.

auto, can only be declared inside a program block, and means that the identifier names an object known only within the program block, less any local redeclarations. That is, it has "block scope". The lifetime of **auto** objects is the time between each entry and exit of the program block, so the value of an **auto** is lost between invocations of the program block that knows about it. Since functions may be called recursively, multiple instances of the same **auto** may exist simultaneously, one instance for each dynamic activation of its program block.

register can only be declared inside a program block, and means much the same as **auto.** That is, it has "block scope".

Two special features of an object declared to be of class **register** are

a) the address of the object cannot be taken, and hence

b) efficient storage, such as the machine's fast registers, can and should be favored to hold the object.

It is not considered an error to declare more objects of class **register** than can be accommodated; excess ones are simply taken as **auto.** The lifetime of **register** objects is the same as **auto** objects. An argument declared to be of class **register** may be copied into a fast register on entry to the function.

typedef means that the name should be recognized as a type specifier, not associated with any object, hence lifetime is irrelevant. Redeclaring a **typedef** in a contained program block is permissible but not advisable. Example of scope rules:

```
extern int i; /* program scope */
static int j;   /* file scope */
extern int f()  /* program scope */
    {
    auto int i; /* new i, block scope */
```

```
    static int j;   /* new j, block scope */
        {
        extern int i;   /* same as first i */
        static int j;   /* even newer j */
        typedef int T;  /* block scope */
        }
    /* T and newer j are gone here */
    }
/* new i and j are gone here */
```

Type

All types in C must be built from a fixed set of basic types: the valueless type **void**; the integer types **char**, **signed char**, **unsigned char**, **signed short**, **unsigned short**, **signed int**, **unsigned int**, **signed long**, and **unsigned long**; and the floating types **float**, **double**, and **long double**.

int may have a representation identical to either **short** or **long**, depending on the target machine. **char** may have a representation identical to either **signed char** or **unsigned char**, depending on the target machine. **short**, **int**, and **long** are signed unless explicitly declared to be unsigned.

From these are derived the composite forms **struct**, **union**, **enum**, **bitfield**, **pointer to**, **array of**, and **function returning**. Recursive application of the rules for deriving composite types leads to a large, if not truly infinite, assortment of types.

void occupies no space and may have no values. It may participate in composite forms only as **pointer to void** or **function returning void**.

signed or unsigned char is a one-byte integer, something just big enough to hold any of the characters in the machine's character set. C promises that printable characters and common whitespace codes are small positive integers other than zero. A **char** can represent at least eight bits of data.

signed or **unsigned short** is a two-byte integer, something just big enough to hold reasonable counts.

signed or unsigned int is either a two-byte or four-byte integer, depending on the target machine. In most microcontroller environments they are two byte integers. An **unsigned int** is guaranteed to be big enough to count all the bytes in the largest object that can be declared.

signed or unsigned long is a four-byte integer, something comfortably large.

float is a floating number of short precision, occupying four bytes.

double is a floating number of longer precision than **float**, occupying eight bytes.

long double is a floating number of precision at least as long as **double**, depending on the target machine. In most microcontroller environments, **long double** is equivalent to **double.** [**Note:** The compiler provides an option that coerces all doubles and long double words to floats. This feature is an extension to the ANSI standard package.]

struct is a sequence of one or more member declarations, aligned well enough and with holes as needed to keep everything on proper storage boundaries for the target machine. There are contexts in which a struct may have unknown content. Members may be any type but **void, function returning, array of unknown size,** and **struct of unknown content.**

union is an alternation of one or more members, the **union** being big enough and aligned well enough to accept any of its member types. Members may be any types but **void, function returning, array of unknown size,** and **struct of unknown content.** A **union of unknown content** is treated just like a **struct of unknown content.**

enum is an integer large enough to represent all of its declared values, each of which is given a name. The names of **enum** values occupy the same space as type definitions, functions, and object names. An **enum of unknown content** is treated just like a **struct of unknown content.**

bitfield is an integer represented as a contiguous subfield of an **int** or **unsigned int,** always declared as a member of **struct.** It participates in expressions much like an **unsigned int,** but its address may not be taken. [**Note:** An extension to the compiler provides the ability to declare a bitfield as an **int** or a **char,** giving the user the ability to mix **char**-based and **int**-based bitfields in the same program.]

pointer to is used to hold the address of some function or object. Pointers may come in different sizes, and have different representations, depending on the type and address space pointed to. A **pointer to void** can hold the address of any object. Pointers to different C objects or functions (of the same type) never compare equal to each other. No C object or function will ever have an address of zero. [**Note:** These comparision rules may not hold for objects created by other languages mixed with C.]

array of is a repetition of some type, the size of which is either a positive compile-time constant or else is unknown. Any type but **void, function returning, array of unknown size,** and **struct of unknown content** may be used in an array.

function returning is a body of executable text, the invocation of which returns the value of some type. A **function returning void** returns no usable value. Any type but **function returning, array,** and **struct of unknown content** may be returned by a function.

Later discussion of statements and expressions will make frequent use of the following type categories:

a) an **integral type** is any of the **basic integer types (char, int** (all sizes, both signed and unsigned)), plus **enum** and **bitfield.**

b) a **floating type** is any of the **basic floating types** (float, double, long double).

c) an **arithmetic type** is an **integral type** or **floating type.**

d) a **scalar type** is an **arithmetic type** or a **pointer to.**

Note that different types are always considered different, for the sake of type checking, even if they have the same target representation. Thus **char** is always a different type from **signed char** or **unsigned char,** and **int** is always different from **short** or **long. int** is, however, a legitimate synonym for **signed int,** as is **short** for **short int** and **long** for **long int.**

Other Name Spaces

struct, union and **enum** tags have scope rules just like object identifiers, except that they form a separate space of names.

Labels in a function body have a scope that extends from first appearance, in a **goto** or as a statement label, through the end of the function body; they may not be redefined within that scope. Labels also form a separate name space.

Members of a **struct** or **union** occupy a separate name space for each **struct** or **union.** [Note: For compatibility with older C programs, different behavior can be obtained as a compile-time option. With this behavior, members of a **struct** or **union** have a scope that extends from first ap-

pearance, in the definition of contents of the **struct** or **union**, through the end of the program file. A member may not be redefined within any **struct** or **union**, unless the new definition calls for the same type and offset.]

Declarations

Declarations form the backbone of a C program. They are used to associate a scope, storage class, and type with function and object identifiers; to name types, enumeration values, and members of **struct** or **union**; to specify the initial values of objects named by identifiers; and to introduce the body of executable text associated with each function name. There are four types of declarations: external, structure, argument, and local. The cast operator and prototype argument declaration may use an abbreviated form of declaration to specify type only.

External Declarations

External declarations have one of the forms:

[<sc>] [<ty>] <decl> [<fn-body>]|
[<sc>][<ty>][<decl> [<dinit>][, <decl> [<dinit>]]*];

i.e., a storage class and type specifier, followed either by a function declaration **<decl>** and an optional function body, or by a comma separated list of declarators **<decl>**, each optionally initialized, the list ending in a semicolon.

Examples are:

```
extern int f(), i, j;  /* declarations only */
static int x = 3, y = 4;  /* objects with initializer */
extern int g()  /* function with body */
     {
     return (3);
     }
```

Storage Class

The storage class <sc> may be **extern**, **static**, or **typedef**; the default is extern.

Type

A type **<ty>** may be qualified by an arbitrary number of type modifiers **<ty-mod>;**

a) **const** means that an object of this type is not modifiable after it is given its initial value, or that a function of this type has no side effects.

b) **volatile** means that an object of this type may be changed unexpectedly by agencies not obvious to the compiler, and hence the compiler should avoid most optimizations.

In addition, Whitesmiths C provides other type modifiers for each microcontroller environment. See your Cross Compiler User's Manual for information about the extensions your compiler supports.

The type modifiers may be intermixed with:

a) a basic type,

b) a **struct** or **union** declaration, described below, or

c) an identifier earlier declared to be a **typedef.**

The default is **int.**

The basic types may be written as:

```
[ signed | unsigned ] char |
[ signed | unsigned ] [ short | long ] [ int ] |
  float |
[ long ] double
```

or any reordering of one of these combinations.

For example, there are nine basic types involving **int:**

```
signed short int              signed long int signed int
unsigned short int            unsigned long int unsigned int
short int      long int       int
```

To declare a pointer to a constant string:

```
const char *p1;
char const *p2;
```

To declare a constant pointer to a string:

```
char * const p;
```

Declarations

A **<decl>** is recursively defined as, in order of decreasing binding:

ident - **ident** is of type **<ty>.**

<decl> '[' **<const>** ']' - where '[' and ']' signify actual brackets. **<decl>** is of type **array of <ty>. <const>** is the unsigned repetition count.

<decl> '[' ']' - where '[' and ']' signify actual brackets. **<decl>** is of type **array of unknown size of <ty>.**

***<ty-mod> <decl>** - **<decl>** is of type **pointer to <ty>. <ty-mod>** is an arbitrary number of type modifiers, described above in the section "Type", which apply to the pointer itself, as opposed to the function or object pointed to.

<decl> ([<id> [, <id>]*]) - **<decl>** is of type **function returning <ty>.** The comma-separated list of identifiers is used only if the declaration is associated with a function body. No constraints are imposed by this form on actual arguments in a subsequent function call.

<decl> (<aty> <a-decl> [, <aty> <a-decl>]* [, ...]) - **<decl>** is of type **function returning <ty>.** The comma-separated list of declarations in parentheses describes the required type of each corresponding actual argument. This form of declaration is called a "function prototype". In a subsequent function call, an actual argument must be assignable to an object of the type declared. If the ellipsis ... is present, additional actual arguments may occur, with no constraints imposed on such arguments. The interpretation of <aty> and **<a-decl>** is the same as for argument declarations (see "Argument Declarations"), except that the identifier may be omitted within any of the **<a-decl>**, as is the case for casts (see **"Casts").** The scope of any identifiers declared within the **<a-decl>** is limited to the parentheses enclosing the list. If the first **<aty>** is **void,** then no additional declarations may follow, and no actual arguments may occur on any subsequent function calls.

(<decl>) - <ty> is redefined, inside the parentheses, as that type obtained for X if the entire declaration were rewritten with (**<decl>**) replaced by X.

This last rule has profound implications. It is intended, along with the rest of the <decl> notation, to permit declarators to be written much as they appear when used in expressions. Thus, * for "pointer to" corresponds to * for "indirect on", () for "function returning" corresponds to () for "called with", and [] for "array of" corresponds to [] for "subscripted with". Declarators must thus be read inside out, in the order in which the operators would be applied.

The following example highlights this critical point:

```
int *fpi(); /* function returning pointer to int */
int (*pfi)();   /* pointer to function returning int */
```

Initializers

There are two basic kinds of initializers, one for objects of type **function returning,** and one for everything else. The former is usually referred to as the definition of a function, i.e., the body of executable text associated with the function name. A **typedef** may not be initialized; a **static** or **extern** may be initialized at most once in the program file. An external declaration with no explicit storage class specifier is taken as **extern,** with the understanding that an initializer of all zeros will be supplied (just as for **static)** if no initializer occurs in the program file. An **extern** must be initialized exactly once among the entire set of files making up a C program. [**Note:** As an extension, an initializer may be omitted in some implementations.]

Data initializers (i.e. **<dinit>)** are described in this section.

Objects may not be declared to be of type **void.** Initialized objects may not be declared to be of type **struct of unknown content.**

Function bodies (i.e. <fn-body>) are described in the section "Statements". For now it will merely be observed that each function body begins with an argument declaration list, and each program block within the function body begins with a local declaration list.

Structure and Enumeration Declarations

If the type specifier in a declaration begins with **struct**, **union**, or **enum**, it must be followed by one of the forms:

```
<tag> { <dlist> } |
  <tag> |
  { <dlist> }
```

The first form defines the content of the structure or enumeration declaration as **<dlist>** and associates the definition with the identifier **<tag>**. The second form can be used to refer either to a **struct of unknown content** or as an abbreviation for an earlier instance of the first form. The last form is used to define content without defining a tag.

By special dispensation, the second form has additional meaning if no declaration list follows. The forms

```
struct <tag>;
union <tag>;
enum <tag>;
```

all indicate that a new instance of **<tag>** is being introduced in the current innermost scope, if none already exists in that scope. This is needed to shield sets of mutually referencing **struct** or **union** declarations from similarly tagged type specifiers in containing scopes. (The usual meaning of such a declaration is that zero objects of the specified type are being declared, i.e. it is normally innocuous.)

Structures and Unions

In a **struct** or **union**, **<dlist>** is a sequence of one or more member declarations of the form:

```
[ <ty> ] <sudecl> [, <sudecl> ]* ;
```

where **<sudecl>** is one of the forms:

```
<decl> [ : <width> ] |
  : <width>
```

<ty> and **<decl>** are the same as for external declarations, except that the types **function returning**, **array of unknown size**, and **struct of unknown content** may not be declared. Type modifiers other than **const** or **volatile** are ignored.

If the type is **int** or **unsigned int**, a bitfield specifier may follow **<decl>**, or stand alone. It consists of a colon **:** followed by a compile time constant giving its width in bits. Adjacent **<sudecl>** declarators with bitfield specifiers are packed, as tightly as possible, into adjacent bitfields in an **unsigned int**; bitfield specifiers that stand alone call for unnamed padding. A new **unsigned int** is begun

a) for the first bitfield specifier in a declaration,

b) for the first bitfield specifier following a non-bitfield specifier,

c) for a bitfield specifier that will not fit in the remaining space in the current **unsigned int**, or

d) for a standalone field specifier, the width of which is zero, e.g., **: 0.**

[Notes: An extension to the compiler provides the ability to declare a bitfield as an **int, unsigned int, char** or **unsigned char.** It is possible to declare **int** based **bitfields** and **char** based **bitfields** in the same unit of compilation.

If this extension is used, adjacent declarations of the same base type **(char** or **int)** are packed as tightly as possible into adjacent fields.

A new memory storage allocation **(char** or **int)** is done

a) for the first bitfield in a declaration

b) for the first bitfield specifier following a non-bitfield specifier,

c) for a change of the base type **(char** to **int** or **int** to **char),**

d) for a bitfield specifier that will not fit in the remaining space in the current based type **(char** or **int),** or

e) for a standalone field specifier, the width of which is zero, e.g., : 0.]

Bitfields by default are packed right to left; i.e. the least significant bit is used first. They can also be packed left to right at the users' option.

For example,

FORTRAN-style complex numbers may be declared as:

```
typedef struct
```

```
    {
    float re, im;
    } COMPLEX;
static COMPLEX c1, c2, *pc;
```

Enumerations

For an **enum**, **<dlist>** is a comma-separated list of one or more member declarations of the form:

<ident> [= <const>]

<ident> is an identifier defined to be an integer constant the value of which is given by the constant expression **<const>**, if specified. Otherwise, the first member of an enumeration declaration is given the value zero and any other member is given a value that is one higher that that of the preceding member.

An example of an **enum** declaration is:

```
enum roman {I=1, II, III, V=5, X=10, L=50, C=100};
```

Argument Declarations

A function initializer begins with an argument declaration list, which is a sequence of zero or more declarations of the form:

[<sc>] [<ty>] <decl> [, <decl>]* ;

The only storage class **<sc>** that may be declared is **register;** the default is a normal argument. **<ty>** and **<decl>** are the same as for external declarations, except that the types **float, char, short, function returning,** and **array of** are misleading if used. On a function call, **float** is coerced to **double,** any integer type shorter than **int** is widened to int, and **function returning** and **array of** become pointers, so any such declaration is a (possibly dangerous) reinterpretation of the actual arguments sent.

The default type for undeclared arguments is **int.**

Examples of function argument declarations are:

```
f(a, b, c)
    double a;
    register int b;
    int c;
    {
    ...
```

}

Local Declarations

Each program block begins with a local declaration list, which is a sequence of zero or more declarations having one of the forms:

<lsc> [<ty>] [<decls>] ; |
[<lsc>] <ty> [<decls>] ;

where **<decls>** is

<decl> [<linit>] [, <decl> [<linit>]]*

In other words, either the storage class **<lsc>** or type **<ty>** must be present.

The storage class **<lsc>** may be **auto, register, extern, static,** or **typedef;** the default is **auto. <ty>** and **<decl>** are the same as for external declarations.

A **static** object declaration may be followed by a data initializer **<linit>,** just like the **<dinit>** used with external declarations, as described in the section "Initializers." An **auto** or **register** may be followed by an **linit** of the form: **assignment operator** =, followed by any expression that may appear in the same context, as the right operand of = with the declared object as the left operand. Such initializers for **auto** and **register** become code which is executed on each entry to the program block.

A register may hold only an object of size **int** (which includes unsigned and pointers to functions and objects that are not **@far).** Anything larger than an **unsigned int** declared to be in a **register** is quietly made an **auto,** as are excess **register** declarations. Anything of class **register** declared smaller than **int** is taken as **register int.**

A local declaration of type **function returning** may not be initialized, i.e. functions cannot be defined inside other functions.

Examples:

```
auto a[10]; /* same as int a[10] */
extern void f();
register int *p = a;     /* p is initialized */
```

Casts

A **cast** is an operator that coerces a value to a specified type. It takes the form:

(<ty> <a-decl>)

<ty> is the same as for external declarations, except that, in conjunction with **<a-decl>**, only **void** or the scalar types (see the section "Type") may be specified. No type modifiers are meaningful in a cast. A **void** cast emphasizes that the value being cast is not to be further used; it is otherwise illegal within a (sub)expression. **<a-decl>** is an abstract declarator, much like the **<decl>** used for external declarations, but with the (unused) identifier optionally omitted. Thus:

```
(int *)   /* coerces to pointer to int */
(struct x *(*)())   /* coerces to pointer to
                       function returning pointer to
                       struct x (!) */
```

To eliminate ambiguity, empty parentheses **( )** are always taken as **function returning,** and never as (unnecessary) parentheses around the omitted identifier.

Redeclaration

Any function or object with program scope or file scope may be redeclared an arbitary number of times provided that all such declarations are compatible.

A redeclaration with storage class **extern** is always compatible with the earlier storage class. A redeclaration with storage class **static** is compatible only with an earlier storage class of **static.**

A redeclaration must have a type identical with the earlier declarations, except that

a) an **array** of unknown size is compatible with a corresponding **array of known size,** the remembered declaration thereafter has a known size for the array; and

b) a **function prototype** is compatible with a corresponding **function returning** having an empty argument list, the remembered declaration thereafter has a prototype for the function.

Thus, information is accured with multiple declarations.

Finally, a redeclaration with no explicit initializer (or function body) is compatible with any earlier initializer. A redeclaration with an explicit initializer is not compatible with any earlier initializers, even if the specified initialization is unchanged.

For example, the following are all compatible:

```
static int a[];
extern int a[5];
static int a[] = {0, 1, 2};  /* padded to 5 elements */
```

Initializers

As part of the declaration process, an object can be given an initial value. This value is established at program startup, for objects with storage class **extern** or **static**, or on each entry to a program block, for objects with storage class **auto** or **register**. If no initializer is specified in a declaration, an **extern** must be initialized in another declaration (not necessarily in the same file), or implicitly initialized to all zeros by an **extern** storage class declaration with the keyword **extern** omitted. [**Note:** For some target environments, such as the MC68HC11, neither an explicit or implicit initializer is required.] A **static** is likewise initialized to all zeros if no initializer is specified, while an **auto** or **register** would contain garbage.

The two basic formats for initializers are:

<decl> = <expr> |
<decl> = <elist>

where **<elist>** is

{ [<expr> | <elist>] [, [<expr> | <elist>]]* [,] }

i.e., a comma-separated list of expressions and lists, with each list enclosed in braces. A trailing comma is explicitly permissible in an **elist**. [**Note:** For compatibility with older C, the = may be omitted; this extension may be disabled.]

auto and **register** declarations with initializers behave much like assignment statements. Only scalar variables may be initialized, but the initializer may be any expression that can appear to the right of an assignment operator when the declared scalar variable is on the left

(see the section "Binary Operators" for information). An **<elist>** is never acceptable in an **auto** or **register** initializer.

Initializers for extern or static Objects

The remaining discussion concerns initializers for objects with storage class **extern** or **static.**

A scalar object is initialized with one **<expr>.** If it is an integral type (see the section "Type"), **<expr>** must be an expression reducible at compile time to an integer literal, i.e., a constant expression, as described in the section "Constants." If the object is a floating type (see the section "Type"), **<expr>** must be an optionally signed floating constant or an integer constant, as an integer constant expression is converted to a floating constant by the compiler. The compiler will not perform even obvious arithmetic involving floating literals, other than to apply unary + or - operators to them.

The form **@<const>** can be used to specify an absolute memory address for a function or object, provided that it has not been previously referenced. This is called an **alias.** No name is published for this form, even for an **extern** declaration. Valid **<const>** values are by nature highly machine-dependent. **[Note:** This is an extension, which may be disabled.] Your Cross Compiler User's Guide provides information on using this extension on your system.

A pointer is initialized with a constant expression, or with the address of an **extern** or **static** object, plus or minus a constant expression. Any constant other than zero is extremely machine-dependent, so this freedom should be exploited only by hardware interface code. If the address of an object appears in a pointer initializer, the object must be previously declared. **[Note:** The compiler can be instructed to accept only zero or pointer expressions of the same type as pointer initializers, as in the proposed ANSI standard.]

A **union** is initialized with one expression; the first member of the union is taken as the object to be initialized.

A **struct** is initialized with either an expression or a list. If an expression is used, the first named member of the **struct** is taken as the object to be initialized. Unnamed padding is implicitly initialized to zeros, in the presence of other initializers for the **struct.** If a list is used, the elements of the list are used to initialize

corresponding named members of the **struct**. If there are more named members than initializers, excess members are initialized with zeros. It is an error for there to be more initializers in a list than named members in the **struct**.

An **array of known size** is initialized much like a **struct**: an expression initializes the first element only, while a list initializes elements starting with the first. If there are more elements than initializers, excess elements are initialized with zeros. It is an error for there to be more initializers in the list than there are elements in the array.

An **array of unknown size**, however, cannot have an excess of initializers, as its multiplicity is determined by the number of initializers provided. After initialization, therefore, an array always has a known size.

By special dispensation, an array of characters may be initialized by a string constant. Therefore:

```
char a[] = "help";
```

is the same as

```
char a[5] = {'h', 'e', 'l', 'p', '\0'};
```

The terminating NUL of a string constant is not used when initializing an **array of known size**.

Any integer, floating, pointer, or string expression may be preceded by a repetition count in square brackets. Thus,

```
[5] 3.0
```

is the same as

```
3.0, 3.0, 3.0, 3.0, 3.0
```

The repetition count must be a constant expression (see "Constants"). A repetition count of zero effectively omits the expression and any trailing comma. [**Note:** Repetition counts are an extension, which may be disabled.]

Elaborate composite types, such as arrays of **structs**, are naturally initialized with lists of sublists, the structure of which reflects the structure of whatever is being initialized. It is often permissible, however, to write an initializer by eliding braces around one or more sublists. In this case, a struct or array (sub)element uses only as many elements as it needs, leaving the rest for subsequent subelements.

In general, it is recommended that complex initializers either have a structure that exactly matches the object to be initialized, or have no internal structure at all. It is hard enough to get either of these extremes correct; intermediate forms frequently defy analysis. The following example shows the initialization of arrays by string constants, as well as the initialization of a complex composite structure.

```
char *p = "help";    /* p points at the string */
char a[] = "help";   /* a contains five chars */
struct complex
    {
    float real, imag;
    } xx[2][3]
    { { {0, 0}, {0, 1}, {0, 2} },
      { {1, 0}, {1, 1}, {1, 2} } };
```

Statements

A C function definition consists of the function declaration proper, followed by any argument declarations, followed by a **<program-block>** which describes the action to be performed when the function is called. A **<program-block>** begins with a **{**, optionally followed by a sequence of local declarations, optionally followed by a sequence of statements, and ends with a **}**. A **<program-block>** may be used recursively whenever a **<statement>** is permitted.

The outermost **<program-block>** differs from a contained **<program-block>** in two ways:

a) any arguments to the function are effectively declared within the **<program-block>**, and

b) a **return** statement effectively follows the **<program-block>**.

The following are the legal **<statement>**s of a C program:

<expression>; - An **<expression>** terminated by a semicolon is a statement that causes the **<expression>** to be evaluated and any result discarded. Assignments and function calls are simply special cases of the expression statement. It is considered an error if **<expression>** produces no useful side effect, i.e., **a = b;** is useful, but **a + b;** is not.

; - A semicolon standing alone is a null statement. It does nothing.

if (<expression>) <statement> [else <statement>] - If the **<expression>** evaluates to a non-zero of any scalar type (see "Type"), the **<statement>** following it is evaluated and the **else** part, if present, is skipped; otherwise the **<statement>** following **<expression>** is skipped and the **else** part, if present, has its **<statement>** evaluated. As in all languages, each **else** part in a nested **if** statement is associated with the innermost "un-**else**'d" **if**.

switch (<expression>) <statement> - If **<expression>**, which must be an integral type (see "Type"), matches the value associated with any of the **case** labels in the statement following, execution resumes immediately following the matching label. If the label **default** is present in the statement following, execution resumes immediately following it. Otherwise execution resumes with the statement following the switch statement. All **case** label values are coerced to the type of **<expression>**. The statement controlled by a switch is typically a **<program-block>**, but doesn't have to be.

case <value> [..<value2>]: <statement> - The **case** statement may only occur within a **switch** statement, as described above. The value must be a constant expression, which is coerced to the type of the **switch** expression; it must not match any other case **<value>**, after coercion, in the same switch. If **..<value2>** is present, the effect is as if case labels were written for the inclusive range of values. [**Note:** Case ranges are an extension, which may be disabled.]

default: <statement> - The **default** statement may only occur within a **switch** statement, as described above. It may occur at most once in any **switch**.

while (<expression>) <statement> - So long as **<expression>** evaluates to a non-zero of any scalar type, **<statement>** is executed. **<expression>** is evaluated prior to each execution of the **<statement>**, plus one more time if it ever evaluates to zero. The **<statement>** may thus be executed zero or more times.

do <statement> while (<expression>); - The **<statement>** is executed and, so long as **<expression>** evaluates to non-zero of any scalar type, the **<statement>** is repeated. The <statement> may be executed one or more times; <expression> is evaluated following each execution of the **<statement>**.

for (<ex1>; <ex2>; <ex3>) <statement> - **<ex1>**, **<ex2>**, and **<ex3>** are all **<expression>**s. **<ex1>** is evaluated

exactly once, then so long as **<ex2>** evaluates to non-zero of any scalar type the **<statement>** is executed and **<ex3>** is evaluated. Thus the **for** behaves much like the sequence **{<ex1>; while (<ex2>) {<statement> <ex3>; }}**

break; - A **break** statement causes immediate exit from the innermost containing **switch, while, do,** or **for** statement, i.e., execution resumes with the statement following. A **break** statement may only occur inside a **switch, while, do,** or **for.**

continue; - A **continue** statement causes immediate exit from the **<statement>** part of the innermost containing **while, do,** or **for** statement, i.e., execution resumes with the "test" part of a **while** or **do,** or with the **<ex3>** part of a **for.** A **continue** statement may only occur inside a **while, do,** or **for.**

goto <identifier>; - A **goto** statement causes execution to resume with the **<statement>** part of a **label** statement having a matching **<identifier>** and contained within a common **<program-block>.** Such a **label** statement must be present.

<identifier>: <statement> - A **label** statement serves as a potential target for a **goto,** as described above. All labels within a given **<program-block>** must be unique **<identifier>s.**

return [<expression>]; - If the **<expression>** is present, it is evaluated and coerced to the type returned by the function, then the function returns with that value. If the **<expression>** is absent, the function returns with an undefined value. A **function returning void** may have no **expression** in any **return** statement. There is an implicit return statement (with no defined value) at the end of each **<program-block>** at the outermost level of a function definition.

Expressions

C offers a rich collection of operators to specify actions on integers, floats, pointers and, occasionally, composite types. Operators can be classified as addressing, unary, or binary. Addressing operators bind most tightly, left to right from the basic term outward; then all unary operators are applied right to left, beginning with (at most one) postfix ++ or --; finally all binary operators are applied, binding either left to right or right to left and on a multi-level scale of precedence.

Parentheses may be used to override the default order of binding, without fear that redundant parentheses will alter the meaning of an expression, i.e., **f(p)** is the same as **f((p))** or **(f(p))**. The language makes few promises about the order of evaluation, however, or even whether certain redundant computations occur at all. Expressions with multiple side effects can thus be fragile, e.g., ***p++ = *++p** can legitimately be evaluated in a number of incompatible ways.

The compiler also reserves the right to regroup commutative operators (such as *, +, |, ^), even in the presence of parentheses. Thus, **(a + b) + c** may be regrouped as **a + (b + c)**. However, C always honors parentheses that follow a unary +. Thus, **+(a + b) + c** will never be regrouped.

Some operands must be in the class of "locators", i.e., things that provide a recipe for locating a function or an object. An identifier declared as a function or object is the simplest locator. The result of the expressions ***p**, **a[i]**, or **p->m** are locators. The result of **x.m** is a locator if **x** is a locator.

Some operands must be in the class of "lvalues", i.e., things that make sense only on the left side of an assignment operator. An lvalue is a locator with a non-**const** type other than **function returning ...** or **array of ....** All scalar and structure expressions also have an "rvalue", i.e., a thing that makes sense only on the right side of an assignment operator. All locators are also rvalues.

Nearly all C operators deal only with scalar types (see "Type"). Where a scalar type is required and a composite type is present, the following implicit coercions are applied: **array of ...** is changed to **pointer to ...** with the same address value; **function returning ...** is changed to **pointer to function returning ...** with the same address value; **struct** or **union** is illegal.

Addressing Operators

func([expr [, expr]*]) - func must evaluate to the type **function returning ...** or **pointer to function returning** and is the function to be called to obtain the rvalue of the expression, which is of type ... Any arguments are evaluated, in unspecified order, and fresh copies of their values are made for each function call (thus the function may freely alter its arguments with limited repercussions). **char** or **short** argument expressions are widened to **int**, and **float** to **double**. All arguments must be scalar or a **struct** or

union. In the absence of a prototype declaration, no checking is made for mismatched arguments or an incorrect number of arguments, but no harm is done providing the highest numbered argument actually used and all preceding arguments correspond properly. In the presence of a prototype declaration, the arguments are coerced as if by assignment. Note that a function declared as returning anything smaller than an **int** actually returns **int**, while a function returning **float** actually returns **double**.

a[i] - is entirely equivalent to ***(a + i)**, so the unary * and binary + should be examined for subtle implications. If **a** is of type **array of ...** and **i** is of type **int**, however, the net effect is to deliver a locator for the **ith** element of **a**, having type ...

x.m - **x** must be of type **struct** or **union** containing a member named **m**. **m** can never be an expression. The object located (or value, if **x** is not a locator) is that of the **m** member of **x**, with type specified by **m**.

p->m - **p** must be of type **pointer to struct** or **union** containing a member named **m**. **m** can never be an expression. The object located is the **m** member of the **struct** or **union** pointed at by **p**, with type specified by **m**.

Unary Operators

***p** - **p** must be of type **pointer to ...** The function or object located is that currently pointed at by **p**, with type ...

&x - **x** must be a locator other than **bitfield** or **register**. The result is an rvalue which is a pointer that points at **x**; the type is **pointer to ...** for **x** of type ...

+x - x must be an arithmetic type (see "Type"). The result is an rvalue of the same value and type as **x**.

-x - **x** must be an arithmetic type. The result is an rvalue which is the negative of **x** and the same type as **x**.

++x - **x** must be a scalar type lvalue. **x** is incremented in place by one, following the rules of addition explained below. The result is an rvalue having the new value and the same type as **x**.

--x - **x** must be a scalar type lvalue. x is decremented in place by one, following the rules of addition explained below. The result is an rvalue having the new value and the same type as **x**.

x++ - **x** must be a scalar type lvalue. **x** is incremented in place by one, following the rules of addition explained below. The result is an rvalue having the old value and the same type as **x**.

x-- - **x** must be a scalar type lvalue. **x** is decremented in place by one, following the rules of addition explained below. The result is an rvalue having the old value and the same type as **x**.

~x - **x** must be an integral type. The result is an rvalue which is the ones complement of **x**, having the same type as **x**.

!x - **x** must be a scalar type. The result is an rvalue which is an integer 1 if **x** is zero; otherwise it is an integer 0.

(<a-type>) x - **<a-type>** is any scalar type declaration (or void) with the identifier optionally omitted, e.g., **(char *)**. The result is an rvalue obtained by coercing **x** to **<a-type>**. This operator is called a "cast", as explained in the section on "Casts."

$noside x - **x** can be any expression. The result is **x** if no side effects accompany its evaluation. Otherwise, an error is reported. Side effects include accessing a **volatile** object, calling a non-**const** function, or using ++, --, or any of the assigning operators.

sizeof x, sizeof (<a-type>) - The result is an integer rvalue equal to the size in bytes of **x** or the size in bytes of an object of type **<a-type>**. **<a-type>** need not be **scalar**, but must not be **void, struct of unknown content, array of unknown size,** or **function returning.**

Binary Operators

There is an implicit "widening" order among the arithmetic types. In general, the type of a binary operator is the wider of the types of its two operands, the narrower operand being implicitly coerced to match the wider. The order of widening, from narrowest to widest, is as follows: **signed char, char, unsigned char, short, unsigned short, int, bitfield, unsigned int, long, unsigned long, float, double,** and **long double. long double** is the widest

type. Type **enum** may be represented as narrower than **int** and is widened to **int**. If arithmetic is not done in place, as in **i += j**, then integer arithmetic is always performed on operands coerced by widening to at least **int**.

[**Note:** An extension to the compiler provides ability to avoid widening operands whenever possible. For example, if i, j, and k are all **chars**, the assignment

```
i = j + k
```

involves only 8-bit arithmetic.]

Coercions are made up from a series of transformations: A **char** or **short** becomes an **int** of the same value. Whether sign extension occurs when widening a plain **char** depends on the flags passed to the parser (**p1**) pass of the compiler. See the description of the **-u** flag to the parser pass in your Cross Compiler User's Guide for information. Sign extension occurs for all other integer types not declared as unsigned; the latter are widened by padding with zeros on the left. An **int** is simply redefined as an **unsigned int** (on twos complement machines at least), with no change in representation. A **bitfield** is unpacked into **unsigned int**. An integer is converted to a floating type of the nearest numerical value, but with possible loss of significant bits on the right. Floating types are converted to wider floating types, seldom with loss of significance.

Assignment may call for a "narrowing" coercion, which is performed by the following operations: A **long double** is either truncated or rounded to its nearest arithmetic equivalent in **double** or **float** format. A **double** is similarly reduced to a **float**. The effect of any overflow is undefined. Conversion to an integer involves discarding any fractional part, then truncating as need be on the left without regard to overflow. Similarly, integers are converted to narrower types by left truncation. The effect of any overflow, when assigning to signed integers, is undefined.

The binary operators are listed below in descending order of binding. Those with the highest precedence appear first. All produce rvalues.

x*y - Both operands must be arithmetic types. The result is the product of x and **y**, with the type of the wider.

x/y - Both operands must be arithmetic types. The result is the quotient of **x** divided by **y**, with the type of the wider. A negative integer quotient may truncate either toward or away from zero, depending on the

machine; positive integer quotients always truncate toward zero. Precedence is the same as for *.

x%y - Both operands must be integer types. The result is the remainder obtained by dividing **x** by **y**, with the type of the wider. Precedence is the same as for *.

x+y - If either operand is of type **pointer to ...**, the other operand must be an integer type, which is first multiplied by the size in bytes of the type **...**, and then added to the pointer to produce a result with the type of **x**; **...** must not be **void, function returning, array of unknown size,** or **struct of unknown content.** Otherwise both operands must be arithmetic, in which case the result is the sum of **x** and **y**, with the type of the wider.

x-y - If **x** is of type **pointer to ...** and **y** is an integer type, **y** is first multiplied by the size in bytes of the type **...** and then subtracted from the pointer to produce a result with type of **x**. The same constraints apply to **...** as for +. If **x** is of type **pointer to ...** and **y** is of the same type (preferably within the same object), then **y** is subtracted from **x** and the result divided by the size in bytes of **...** to produce an integer result. Otherwise both **x** and **y** must be arithmetic types; the result is **y** subtracted from **x**, with the type of the wider. Precedence is the same as for +.

x<<y - Both operands must be integer types, The result is **x** left shifted **y** places, with the type of **x**. No promises are made about the value of such an expression if **y** is larger than the number of bits in **x**, or negative.

x>>y - Both operands must be integer types. The result is **x** right shifted **y** places, with the type of **x**. If the result type is **unsigned,** no sign extension occurs on the shift. If it is **signed,** sign extension does occur. No promises are made if **y** is larger than the number of bits in x or negative. Precedence is the same as for <<.

x<y, x<=y, x>y, x>=y - If either operand is of type **pointer to ...** and the other is of an integer type, the integer is not scaled. If both operands are of type **pointer to ...** the pointers are compared for order within their address space. Otherwise both **x** and **y** must be arithmetic types, and the narrower is widened to match the type of the wider before the comparison is made. The result is an integer 1 if the relation obtains; otherwise it is an integer 0.

x==y, x!=y - The operands are coerced in the same way as for <, and then compared for equality (==) or inequality (!=). The result is an integer 1 if the relation obtains; otherwise it is an integer 0.

x&y - Both operands must be integer types. The result is the bitwise **and** of **x** and **y**, with the type of the wider.

x^y - Both operands must be integer types. The result is the bitwise **exclusive or** of **x** and **y**, with the type of the wider.

x|y - Both operands must be integer types. The result is the bitwise **inclusive or** of **x** and **y**, with the type of the wider.

x&&y - Both operands must be scalar types. If **x** is zero, the result is taken as integer 0 without evaluating **y**. If **x** is nonzero then **y** is evaluated. The result is integer 1 only if both **x** and **y** are nonzero.

x||y - Both operands must be scalar types. If **x** is nonzero, the result is taken as integer 1 without evaluating **y**. The result is integer 1 if either **x** or **y** is nonzero. If both **x** and **y** are zero, the result is integer 0.

t?x:y - If **t**, which must be a scalar type, is nonzero, the result is **x** coerced to the final type. Otherwise the result is **y** coerced to the final type. Exactly one of the two operands **x** and **y** is evaluated. The final type is **pointer to ...** if either operand is **pointer to ...** and the other is an integer type (the integer is not scaled). If both operands are of the same type **pointer to ...** the final type is the same. If both operands are arithmetic types, the final type is the wider of the two types. Otherwise both operands must be the same **struct** or **union** type, which is the type of the result.

x=y - Both operands must be scalar types, or else both must be of the same **struct** or **union** type. **x** must be an lvalue. y is coerced to the type of **x** and assigned to x. If x is a **pointer to ...**, **y** may be the same **pointer to ...** or an integer type (the integer is not scaled). If either operand is a **pointer to void**, the other may be any **pointer to ...** Otherwise both operands must be of arithmetic type. The result is an rvalue equal to the value just assigned, having the type x.

x*=y, x/=y, x%=y, x+=y, x-=y, x<<=y, x>>=y, x&=y, x^=y, x|=y - Each of the operations **x op= y** is equivalent to x =

x op y, except that **x** is evaluated only once and the type of **x op y** must be that of **x**, e.g., **x -= y** cannot be used if **x** and **y** are both pointers. **[Note:** For historical reasons, the operators may also be written **=op**. This extension may be disabled.] Precedence is the same as for **=**.

x,y - **x** is evaluated first, then **y**. The result is the value and type of **y**. Note that commas in an argument list to a function call are taken as argument separators, not comma operators. Thus **f(a,b,c)** represents three arguments, while **f(a,(b,c))** represents two, the second one being **c** (after **b** has been evaluated).

Constants

There are several contexts in a C program where expressions must be in the proper form for evaluation at compile time, such as the expression part of a **#if** preprocessor control line, the size of an array in a declaration, the width of a **bitfield** in a **struct** declaration, and the label value of a **case** statement. In many other contexts the compiler attempts to reduce expressions, but this is not mandatory except in the interest of efficiency.

The **#if** statement evaluates expressions using **long** integer arithmetic. No assignment operators, casts, or **sizeof** operators may be used. The result is compared against zero. There is no guarantee that large numbers will be treated the same across systems, due to the variation in operand size, but this variation is expected to be minimal among **longs.**

Bitfield widths, array sizes and case labels are also computed using **long** integer arithmetic. The **sizeof** operator and casts are permitted in such expressions.

In all other expressions, the compiler applies a number of reduction rules to simplify expressions at compile time. These include the following (assumed) identities:

```
x * 1 == x
x / 1 == x
x + 0 == x
x - 0 == x
x + y == y + x
(x + y) + z == x + (y + z)
x && true == x
x || false == x
false && x == false
```

```
true || x == true
(x + y) * z == x * z + y * z
```

plus a number of others. In other words, certain subexpressions may generate no code at all, if the operation is patently redundant. This is why the **volatile** type modifier is an important attribute for objects used in writing I/O drivers and other machine dependent routines that are expected to produce useful side effects not obvious to the compiler. The compiler avoids reductions that would change the pattern of accesses to a **volatile** object.

The compiler does not currently perform common subexpression elimination, however, nor does it rearrange the order of computation between statements. This insures at least a minimal amount of determinism.

To ensure that compile time reductions occur, on the other hand, it is best to group constant terms within an expression so the compiler does not have to guess the proper rearrangement to bring constants together.

The Preprocessor

A preprocessor is used by the C compiler to perform **#define, #include,** and other functions signalled by the # control character, before actual compilation begins. A number of options can be specified at compile time by the use of command line options. Some of these optionss and their effects are mentioned below. See your Cross Compiler User's Guide for more information.

Comments (i.e. anything appearing between the delimiters **/*** and ***/**) are effectively replaced by a single space, and any line that ends with a **newline** is merged with the line following it. If the first non-whitespace character on the resultant line matches the preprocessor control character # (or the secondary preprocessor control character, which may be specified at compile time), the line is taken as a command to the preprocessor. All other lines are either skipped or expanded, as described below.

The following command lines are recognized by the preprocessor:

#define <ident> <defn> - defines the identifier **<ident>** to be the definition string **<defn>** that occupies the remainder of the line. Identifiers consist of one or more letters, digits, and underscores _, where the first character is a non-digit; the first 63 charac-

ters are used for comparing identifiers. A sequence of zero or more formal parameters, separated by commas and enclosed in parentheses, may be specified, provided that no whitespace occurs between the identifier and the opening parenthesis. The definition string begins with the first non-whitespace character following the identifier or its parameter list, and ends with the last non-whitespace character on the line. It may be empty. An identifier may be redefined only if the new definition exactly matches the older one.

#undef <ident> - removes the definition for **<ident>**, if any. It is not considered an error to **#undef** an undefined identifier.

#include <fname> - causes the contents of the file specified by the filename **<fname>** to be lexically included in place of the command line. Macro expansion occurs before **<fname>** is examined. A filename can be an arbitrary string inside (literal) quotes "" or an arbitrary string inside (literal) angle brackets < >. In the latter case, a series of standard prefixes is prepended to the filename (normally just the null prefix "" unless otherwise specified at invocation time), to locate the file in one of several places. If the filename inside the quotes "" cannot be opened for reading, it is treated just like a filename inside angle brackets < >. Included files may contain further **#includes.**

#ifdef <ident> - commences skipping lines if the identifier **<ident>** is not defined. If **<ident>** is defined, processing proceeds normally for the range of control of the **#ifdef.** The range of control is up to and including a balancing **#endif** command. A **#else** command encountered in the range of control of the **#ifdef** causes skipping to cease if it was in effect, or normal processing to change to skipping if skipping was not in effect. A **#elif <expr>** encountered in the range of the #ifdef causes skipping to cease if skipping was in effect, and if no prior #if or #elif in the same group evaluated to non-zero, and if **<expr>** evaluates to non-zero; otherwise skipping continues or commences. It is an error for a **#elif** to follow a #else in a #if group, or for a **#else** or **#elif** to occur outside of a **#if** group. It is permissible to nest #ifdef and other #if groups; entire groups will be skipped if skipping is in effect at the start. Preprocessor commands such as **#define** and **#include** are not performed while skipping.

#ifndef <ident> - is the same as **#ifdef,** except that #ifndef commences skipping if the identifier is defined,

as opposed to undefined.

#if <expression> - is the same as **#ifdef**, except that the rest of the line is first expanded, and then evaluated as a constant expression. Skipping commences if the expression evaluates to zero. An expression may contain parentheses, the unary operators **defined**, +, -, **!**, and **~**, the binary operators +, -, *, /, %, &, |, ^, <<, >>, <, ==, >, <=, =>, !=, && and ||, and the ternary operator ? :. The definitions and bindings of the operators match those for the C language, subject to the constraint that only integer constants may be used as operands. The special unary operator **defined** yields the value 1 if the identifier following it has a macro definition. Otherwise its value is 0.

#line= [<fname>]:<num> - causes the line number to be **<num>** and the file name to be **<fname>** if present. This form is for internal use only.

#line <num> [<fname>] - causes the line number used for diagnostic printouts to be set to **num** and the corresponding filename to be set to the quoted string **fname**, if present. If no filename is specified, the filename used for diagnostic printouts is left unchanged. **num** must be a decimal integer.

#pragma <stuff> - causes **<stuff>** to be passed on to the parsing phase of compilation (the **p1** pass) as "pragmatic" commentary.

- is taken as an innocuous line, if empty. Anything else not recognizably a command causes a diagnostic.

Expansion of non-command lines causes each defined identifier to be replaced by its definition string, and then rescanned for further expansion. If the definition has formal parameters, and the next token is a left parenthesis, then a group of actual parameters, inside balanced parentheses, must occur within a group of lines containing no command lines. The group may be no longer than 512 characters total.

There must be the same number of formal parameters as there are actual parameters. If a formal parameter is followed by a #, however, as in

```
#define x(a, b#, c) ...
```

then actual parameters need not be present from that point onward. Moreover, the last formal parameter matches zero or more parameters not earlier matched. If the last formal parameter is preceded by the #, a non-null actual

parameter is preceded by a comma when expanded. Formal parameters with no actual parameters expand to null token sequences. [**Note:** The use of # is an extension.]

Note that no attempt is made to add whitespace before or after replacement text to avoid blurring of token boundaries, just as no parentheses are added to avoid bizarre arithmetic binding in expressions. No expansion occurs within " ", comments, or ' ' strings.

Builtin Macros

The following identifiers behave much like **#define'd** identifiers, except that they do not test as defined by **#ifdef, #ifndef,** or the **defined** unary operators:

__FILE__ - expands to the name of the current source file, written inside quotes "".

__LINE__ - expands to a decimal number which is the current source line number within the current source file. The first line of the file is number 1.

$if(<expression> [, <args>]) - evaluates **<expression>** just as **#if** does. If the result is non-zero, **$if** expands to **<args>** (including any commas). Otherwise it expands to nothing.

If a definition is provided by **#define,** it supersedes the builtin meaning.

Pragmas

The following pragmas are currently defined:

#pragma debug [on/off] - turns debugging instrumentation on or off, from this point onward, assuming debugging was requested on the parser (**p1**) command line. If neither **on** or off is specified, **on** is assumed.

#pragma space <sc> <class> <names> - sets the default space modifiers for various kinds of declarations, **<sc>** may be one of: **extern, static, auto,** or *, where **extern** represents all identifiers with program scope, **static** represent all indentifiers with file scope, **auto** represents all autos and arguments to functions, and * represents all things pointed to; if more are specified, then all four are affected uniformly. **<class>** may be **()** for functions or **[]** for objects; if neither is specified, then both are af-

fected uniformly. **<names>** may be zero or more space names of the form **@far** or **@<ident>**; if more are specified then no special space information is provided by default.

Thus, for example, a target machine with pointers uniformly larger than **int** can be specified by **#pragma space * @far** and a machine which requires large pointers only for functions can be specified by **#pragma space () @far**. See your Cross Compiler User's Guide for information on using **pragmas**.

Type Modifiers for Access Control

The type modifiers make statements about how objects are accessed. These modifiers are:

const - assures that no valid program may attempt to modify the declared object. Hence it may be placed in Read Only Memory (ROM).

volatile - warns that agencies not obvious to the compiler may be altering the value of the declared object. Optimizations that alter the pattern of accesses should therefore be avoided.

Note that all four combinations of these two adjectives are useful. An object declared as neither **const** or **volatile** is an ordinary (optimizable) C object. The **const** modifier alone signifies a read-only object. The **volatile** type modifier used alone creates a hardware control register, an object in memory shared with other processes, or perhaps just an object touched by interrupt handling functions. An **const volatile** object could be a read-only status register, or a status variable controlled by other agencies.

CHAPTER 3

USAGE CONSIDERATIONS

This chapter describes Whitesmiths' suggestions for good C coding style and explains some of the differences among C compilers that may affect your coding conventions.

Style Tips

C is too expressive a language to be used without discipline; it can rival APL in opacity or PL/1 in variety. The discussion that follows concerns practices and restraints currently in use by Whitesmiths, Ltd. (and throughout the C community) that are recommended for writing good C code.

Commenting

Each file deserves an opening comment, typically in upper-case, indicating the contribution the file makes towards the project. This is important during the maintenance cycle to help determine where a group of changes should be localized. Explanatory comments should precede each group of data and each function body, with a single blank line preceding the comment. If the body of a function is sufficiently complex, a good explanatory comment is the half dozen or so lines of pseudocode that best summarize the algorithm. Additional comments are most effective when placed to the right of the line being explained, separated by one tab stop from the end of the statement.

Organization and Header Files

Related declarations should be packaged together as much as possible, with as many of these declared **static** (LOCAL) as possible. Common definitions, type declarations, and prototypes for all external functions should be grouped into one or more header files to be **#included** as needed with each file.

If any use is made of the ANSI library, its definitions should be included as well, as in:

```
/*  GENERAL HEADER FOR FILE
 *  copyright (c) 1978, 1988 by Whitesmiths, Ltd.
 */
#include <wslxa.h>
#include <stdio.h>

/*  defs.h contains project-wide typedefs,
 *  constants, and function prototypes
 */
#include "defs.h"

#define MAXN    100 /* definitions local to this file */
```

The header file **<wslxa.h>** contains definitions of common constants and portable types (such as LOCAL and IMPORT). These created types and type modifiers are listed in parentheses in this section, next to the normal C types.

Header files should contain **#define**'s, **typedef**'s, and declarations that must be known to all source files making up a program. They should not include any initializers, as these would be repeated by inclusion in multiple source modules and create "multiply defined" messages at link time. A good convention is to use all uppercase characters for **#define**'d identifiers and created typedefs, as a warning to the reader that the language is being extended.

It is also good practice to explicitly import all external references needed in each function body by using **extern** (IMPORT) declarations. This not only documents any pathological connections, but also permits data initializers to be moved freely among files without creating problems.

Within a file, a good discipline is to put all data declarations first, followed by all function bodies in alphabetical order by function name. Data declarations are typically clumped into logical groups, e.g., all flags, all file control information, etc.

If the types provided in <wslxa.h> are not sufficient to describe all the objects used in a program, then all other types needed should be provided by **#define**'s or **typedef**'s. All declarations should be typed, preferably with these defined types, to improve readability.

Restrictions

If a C program totals more than about five hundred lines of code, it should be split into files each no bigger than five hundred lines. This tends to minimize editor startup time and compile time for useful turnaround in the face of many last-minute changes.

The **goto** statement should never be used. The only case that can be made for it is to implement a multi-level break, which is not provided in C, but this seldom proves to be a prudent thing to do in the long run. If a function has no **goto** statements, it has no need for labels.

Other constructs to avoid are the **do-while** statement; which inevitably can be evolved into a safer **while** or **for** statement, and the **continue** statement, which is typically just a way of avoiding the proper use of **else** clauses inside loops.

More than five levels of indenting (see the section "Formatting" below) is a sure sign that a subfunction should be split out, as is also the case with a function body that goes much over a page of listing or requires more than half a dozen local variables. Naturally there are exceptions to all these guidelines, but they are few and far between.

The use of the quasi-Boolean operators **&&**, **||**, **!**, etc. to produce integer ones and zeros should not be indulged to perform cute arithmetic, as in

```
sum[i] = a[i] + b[i] + (10 <= sum[i - 1]);
```

Such practices, if used, should be commented, as should most hard-to-follow bit manipulations using **&**, **|** and **^**.

Parenthesis and Comparisons

Parentheses should be used whenever there is a hint of ambiguity in the order of evaluation. Note in particular that & and | mix poorly with the relational operators, that the assigning operators are weaker than **&&** and **||**, and that << and >> are impossible to guess right. The worst offender is:

```
if ((a & 030) != 030)
    ...
```

which does entirely the wrong thing if the parentheses are

omitted.

Elaborate expressions involving ? and :, particularly multiple instances of them, are often hard to read. Parenthesizing helps, but excessive use of parentheses is just as bad.

If the relational operators > and >= are avoided, then compound tests can be made to read like intervals along the number axis, as in

```
if ('0' <= c && c <= '9')
    ...
```

which is demonstrably true when c is a digit.

Formatting

While it may seem a trivial matter, the formatting of a C program can make all the difference between correct comprehension and repeated error. To get maximum benefit from support tools such as editors and cross referencers, one should apply formatting rules rigorously.

Each external declaration should begin (with optional storage class and mandatory type) at the beginning of a line, immediately following its explanatory comment. All defining material, data initializers or function definitions, should be indented at least one tab stop, with additional tab stops to reflect substructure. Tabs should be set uniformly every four to eight columns.

A function body, for instance, always looks like:

```
TYPE name(arg1, arg2, arg3)
    TYPE1 arg1;
    TYPE2 arg2, arg3;
    {
    <local declarations>

    <statements>
    }
```

This example assumes that **arg2** and **arg3** have the same type. If no **<local declarations>** are present, there is no empty line before <statements>.

<local declarations> consists of first **extern** (IMPORT), then **register** (FAST), then **auto** (no storage class specifier), then **static** (INTERN) declarations; these are sorted alphabetically by type within storage class and alphabetically by name within type. Comma-separated lists

may be used, so long as there are no initializers; an initialized variable should stand alone with its initializer. FAST and **auto** storage should be initialized at declaration time only if **const** (RDONLY), i.e. the value is not expected to change throughout the declared scope.

<statements> are formatted to emphasize control structure, according to the following basic patterns:

```
if (test)
    <statement>
if (test)
    {
    <statement>
    <statement>
    ...
    }
else
    <statement>
if (test1)
    <statement>
else if (test2)
    <statement>
else if (test3)
    ...
else
    <default statement>
switch (value)
    {
case A:
case B:
    <statement>
    ...
    break;
case C:
    <statement>
    ...
    break;
default:
    <statement>
    ...
    }
while (test)
    <statement>
for (init; test; incr)
    <statement>
for (; test; incr)
    <statement>
for (init; ; )
    <statement>
FOREVER   /* instead of for(;;) */
    <statement>
return (expr);
```

Note that, with the explicit exception of the **else-if** chain, each subordinate <statement> is indented one tab stop further to the right than its controlling statement. Without this exception, an **else-if** chain would be written:

```
if (test 1)
    <statement>
else
    if (test 2)
        <statement>
    else
        if (test 3)
            <statement>
        ...
            else
                <default statement>
```

Within statements, there should be no empty lines, or any tabs or multiple spaces embedded in a line. Each keyword should be followed by a single space, and each binary operator should have a single space on each side. No spaces should separate casts, unary or addressing operators from their operands. A possible exception to the operator rule is a composite constant, such as (GREEN|BLUE).

If an expression is too long to fit on a line (of no more than 79 characters) it should be continued on the next line, indented one tab stop further than its start. A good rule is to continue only inside parentheses, or with a trailing operator on the preceding line, so that displaced fragments are more certain to cause diagnostics.

A typical library function, formatted with these rules in mind, looks like this:

```
/*  CONVERT LONG TO BUFFER
 *  copyright (c) 1988 by Whitesmiths, Ltd.
 */
#include <wslxa.h>
#include <baseli.h>

BYTES ltob(is, ln, base)
    FAST TEXT *is;
    LONG ln;
    COUNT base;
    {
    FAST TEXT *s;
    LONG lb;

    s = is;
    if (ln < 0 && base <= 0)
        {
        ln = -ln;
```

```
        *s++ = '-';
        }
    if (base == 0)
        base = 10;
    else if (base < 0)
        base = -base;
    lb = base;
    if (ln < 0 || lb <= ln)
        s += _ltob(s, (ULONG)ln / lb, base);
    *s = "0123456789abcdef"[(ULONG)ln % lb];
    return (s - is + 1);
    }
```

Differences Among C Compilers

The ANSI draft standard is still in flux at this time, although it has come together in many important respects. There are also many other C compilers available besides those created by Whitesmiths, Ltd., and these vary considerably in some areas. This section concerns some of the major dialect problems to watch out for.

When moving from a Whitesmiths C compiler:

- The only base types supported more or less universally are **char**, **short**, **int**, **unsigned int**, **long**, **float**, and **double**.
- The backslash character \ is used to continue lines only within strings in many environments.
- Early compilers have only one name space for all **struct** and **union** members.
- Many compilers do not permit initialization of **bitfields** and **unions**.
- Enumerations, **void**, structure assignment, and functions returning structures are moderately new concepts.
- Prototype declarations, **#elif**, and **long double** are very new concepts.
- The "address of" operator & is often forbidden to operate on arrays and functions.
- Any language feature described in a bracketed note as a language extension is almost certainly not in other C implementations.

Usage Considerations

When moving to a Whitesmiths C compiler:

- Formal arguments narrower than **int** are actually narrowed on most other compilers. You can either redeclare them explicitly as **int**, or use the extension letting the Whitesmiths compiler treat narrow arguments within their own type. Check your Cross Compiler User's Guide for that option. Pay particular attention to any narrow argument of which the address is taken.

- Register variables narrower than **int** are sometimes represented as the narrower type. Redeclare them explicitly as **int** before moving.

- Some compilers do not require any defining instance of an object (either an initial value or the absence of the keyword **extern**). Some versions of Whitesmiths C do require a defining instance. If they are reported as undefined at link time, add = 0 to one declaration.

System Dependencies

Since this implementation produces assembler code for the target system, there is some variation in naming caused by assembler limitations. There may be as few as six, or over sixty, significant characters in external identifiers. Often, only one case of letters is significant.

INDEX

Index

A

Index

Index

E

F

G

Index

H

I

K

L

Index

Index

Q

R

S

Index

T

U

V

W

Index