COSMIC Software

C Language manual

Rev. 2.1

Copyright © COSMIC Software 1999, 2012
All rights reserved.

Table of Contents

Preface

Chapter 1
Historical Introduction

Chapter 2
C Language Overview

C Files 5
    Lines 5
    Comments 6
    Trigraphs 6
Lexical Tokens 7
    Identifiers 7
    Keywords 7
    Constants 8
    Operators and Punctuators 8
Declarations 9
    Integer Types 9
    Bit Type 11
    Real Types 11
    Fractional Types 12
    Pointers 12
    Arrays 13
    Structures 13
    Unions 14
    Enumerations 14
    Functions 14

Chapter 3
Declarations

Integers 18
Reals 18
Fractionals 18
Bits 19
Pointers 19
Arrays 20
Modifiers 21
Structures 24
Unions 27
Enumerations 27

Functions 28
Storage Class 30
Typedef 32
Variable Scope 32
Absolute Addressing 33

Chapter 4
Expressions

Variables 36
Constants 36
Strings 40
Sizeof 41
Operators 42
    Arithmetic operators 43
    Bitwise operators 44
    Boolean operators 45
    Assignment operators 46
    Addressing operators 48
    Function call operator 49
    Conditional operator 51
    Sequence operator 52
    Conversion operator 52
Priorities 54

Chapter 5
Statements

Block statement 60
Expression statement 60
If statement 61
While statement 63
Do statement 64
For statement 65
Break statement 67
Continue statement 68
Switch statement 69
Goto statement 71
Return statement 72

Chapter 6
Preprocessor

Macro Directives 76
    Hazardous Behaviours 80

Predefined Symbols 82
Conditional Directives 84
Control Directives 86
#include 86
#error 86
#line 86
#pragma 87
Space Definition 87
Section Redefinition 88
Loop Unrolling 90
Space modification 90
Inline Assembler 91

Chapter A
Implementation Defined Behavior

Description of Implementation-Defined behavior 94
Translation 94
Environment 94
Identifiers 94
Characters 94
Integers 95
Floating point 95
Arrays and Pointers 95
Registers 96
Structures, Unions, Enumerations, and Bit-fields 96
Qualifiers 96
Declarators 96
Statements 96
Preprocessing Directives 96

Chapter B
Supported C99 Extensions

Description of Supported C99 Extensions 100
Trailing comma in enumeration declaration 100
Flexible array 100
Repeated qualifiers 100
Inline keyword 100
Variable macro arguments 100

Preface

Chapter 1, “*Historical Introduction*”

Chapter 2, “*C Language Overview*”
- file structure, separate compile
- character set (trigraph, logical lines, preprocessor)
- naming, identifiers, keywords, operators, punctuation
- C objects (simple, aggregate)
- functions

Chapter 3, “*Declarations*”
- syntax for each kind of object
- modifiers const, volatile
- extra modifiers for pointers
- function declaration, arguments and local variables
- classes and scope
- type redeclaration (typedef)

Chapter 4, “*Expressions*”
- identifiers, operators and constants
- operators behaviour
- conversions, explicit, implicit

Chapter 5, “*Statements*”
- description of each statement

Chapter 6, “*Preprocessor*”
- description of each directives
- traps and solutions
- pragmas and their usage

Appendix A, “*Implementation Defined Behavior*”
- description of Implementation-Defined behavior

Appendix B, “*Supported C99 Extensions*”
- description of supported C99 extensions

CHAPTER 1

Historical Introduction

The C language was designed in 1972 at the Bell Lab's by Denis Ritchie, mainly to rewrite the UNIX operating system in a portable way. UNIX was written originally in assembler and had to be completely rewritten for each new machine. The C language was based on previous languages named B and BCPL, and is logically called **C** to denote the evolution.

The language is described in the well known book "*The C Programming Language*" published in 1978, whose "*Appendix A*" has been used for many years as the *de facto* standard. Because of the growing popularity (The popularity) of UNIX and of the C language (growing), several companies started to provide C compilers outside of the UNIX world, and for a wide range of processors, microprocessors, and even micro-controllers.

The success of C for those small processors is due to the fact that it is not a **high level** language, as PASCAL or ADA, but a *highly portable macro assembler*, allowing with the same flexibility as assembler almost the same efficiency.

The C language has been normalized from 1983 to 1990 and is now an ANSI/ISO standard. It implements most of the usefull extensions added to the original language in the previous decade, and a few extra features allowing a more secured behaviour.

The ANSI/ISO standard distinguishes two different environments:

- A *hosted* environment describing *native* compilers, whose target processor uses a specific operating system providing disks file handling, and execution environments.
- A *freestanding* environment describing a *cross* compiler, whose target processor is generally an *embedded* microprocessor or a micro-controller without any operating system services. In other words, everything has to be written to perform the simplest operation.

On a *native* system, the same machine is used to create, compile and execute the C program. On an *embedded* system, the target application has no disk, no operating system, no editor and cannot be used to create and compile the C program. Another machine has to be used to create and compile the application program, and then the result must be transferred and executed by the target system. The machine used to create and compile the application is the *Host* system, and the machine used to execute the application is the *Target* system.

COSMIC compilers are *Cross* compilers and conform to the *Freestanding* environment of the ANSI/ISO standard. They also implement extensions to the standard to allow a better efficiency for each specific target processor.

An evolution from the older C89 standard was introduced in 1999. Referred to as C99 (or C9X), the new standard has added a few new types and constructs. Some of these features have been used in COSMIC compilers as C99 conformant extensions to the C89 standard (see **Appendix B,** “*Supported C99 Extensions*”).

CHAPTER
2

C Language Overview

A C program is generally split in to several files, each containing a part of the text describing the full application. Some parts may be already written and used from libraries. Some parts may also be written in assembler where the C compiler is not efficient enough, or does not allow a direct access to some specific resources of the target processor.

C Files

Each of these C files has to be compiled, which translates the C text file to a relocatable object file. Once all the files are compiled, the linker is used to bind together all the object files and the required libraries, in order to produce an executable file. Note that this result file is still in an internal format and cannot be directly transferred to the target system. A special translator is provided to convert the executable file to a downloadable format.

Lines

Each C text file contains a set of lines. A line contains characters and is finished by a line terminator (line feed, carriage return). The C compiler allows several physical lines to be concatenated in a single logical line, whose length should not exceed 511 characters in order to strictly comply with the ANSI standard. The COSMIC compiler accepts up to 4095 characters in a logical line. Two physical lines are concatenated into a single logical line if the first line ends with a backslash character '`\`'

just before the line terminator. This feature is important as the C language implements special directives, known as preprocessing directives, whose operands have to be located on the same logical line.

Comments

Comments are part of the text which are not meaningful for the compiler, but very important for the program readability and understanding. They are removed from the original text and replaced by a single whitespace character. A comment starts with the sequence `/*` and ends with the sequence `*/`. A comment may span over several lines but nesting comments is not allowed. As an extension from the ANSI standard, the compiler also accepts C++ style comments, starting with the sequence `//` and ending at the end of the same logical line.

Trigraphs

The C language uses almost all the ASCII character set to build the language components. Some terminals or workstations cannot display the full ASCII set, and they need a specific mechanism to have access to all the needed characters. These special characters are encoded using special sequences called **trigraphs**. A *trigraph* is a sequence of three characters beginning with two question marks `??` and followed by a common character. These three characters are equivalent to a single one from the following table:

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

All other sequences beginning with two question marks are left unchanged.

Lexical Tokens

Characters on a logical line are grouped together to form lexical tokens. These tokens are the basic entities of the language and consist of:

identifiers
keywords
constants
operators
punctuation

Identifiers

An **identifier** is used to give a name to an object. It begins with a letter or the underscore character **`_`**, and is followed by any letter, digit or underscore character. Uppercase and lowercase letters are not equivalent in C, so the two identifiers **`VAR1`** and **`var1`** do not describe the same object. An identifier may have up to 255 characters. All the characters are significant for name comparisons.

Keywords

A **keyword** is a reserved identifier used by the language to describe a special feature. It is used in declarations to describe the basic type of an object, or in a function body to describe the statements executed.

A keyword name cannot be used as an object name. All C keywords are lowercase names, so because lowercase and uppercase letters are different, the uppercase version of a keyword is available as an identifier although this is probably not a good programming practice.

The C keywords are:

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

Constants

A **constant** is used to describe a numerical value, or a character string. Numerical constants may be expressed as real constants, integer constants or character constants. Integer constants may be expressed in decimal, octal or hexadecimal base. The syntax for constants is explained in the *Expressions* chapter.

Operators and Punctuators

An **operator** is used to describe an operation applied to one or several objects. It is mainly meaningful in expressions, but also in declarations. It is generally a short sequence using non alphanumeric characters. A punctuator is used to separate or terminate a list of elements.

C operators and punctuators are:

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

Note that some sequences are used as operators and as punctuators, such as `*`, `=`, `:`, `#` and `,`.

Several punctuators have to be used by pairs, such as `( )`, `[ ]`, `{ }`.

When parsing the input text, the compiler tries to build the longest sequence as possible for a token, so when parsing:

```
a+++++b
```

the compiler will recognize:

`a ++ ++ + b` which is not a valid construct

and not:

`a ++ + ++ b` which may be valid.

Declarations

A C program is a set of tokens defining objects or variables, and functions to operate on these variables. The C language uses a *declaration* to associate a *type* to a *name*. A type may be a *simple* type or a *complex* type.

A simple type is numerical, and may be integer or real.

Integer Types

An integer type is one of:

`char`	1 byte
`short`	2 bytes
`int`	2 or 4 bytes
`long`	4 bytes
`long long`	8 bytes

An object of a given type will occupy the corresponding size in memory, and will be able to hold a different range of values. Note that these sizes are not defined exactly like that in the ANSI standard, but the shown values are those commonly used with microprocessors. Note that **short** may be written **short int** and that **long** may be written **long int**.

The type **int** is equivalent either to type **short** or to type **long** depending on the target processor capabilities. For most of the existing microprocessors, *int* is equivalent to *short* because their internal registers are at most 16 bits wide. The type *int* is an important one because it is used as a reference type for the expressions and for the function arguments. It is a source of problems when adapting a program to another target processor because its size may also change.

The type **long long** is a C99 extension and is not supported for all the target processors. The compiler provides a predefined preprocessor symbol allowing to check if this type is supported or not (see **__LGLG__** in *page 83*). When supported, this type cannot be used for a bitfield, an enumeration, or a switch expression, as limitations from the standard C99 behaviour.

An integer type may be prefixed by the keyword **signed** or **unsigned** to give a more accurate definition of the range of possible values. The keyword *signed* means that the values hold by the variable may be positive or negative. For instance, a *signed char* will hold values from **-128** to **+127** if numbers are represented using the two's complement convention. This convention is used by all the common microprocessors. The keyword *unsigned* means that the value held by the variable is positive only. An *unsigned char* will hold values from **0** to **255**.

Note that these attributes do not change the object size in memory, but alter the way the object is handled. For instance, comparing an *unsigned integer* less than zero is meaningless, and will probably lead in a compiler error message. Other effects of these attributes will be discussed with the expressions and conversions.

If these keywords are not used in an integer declaration, the object has a default attribute depending on its type.

`short`	are signed by default
`int`	
`long`	
`long long`	
`char`	is implementation dependant

A plain char object may be either *signed* or *unsigned*, depending on which attribute is simpler to implement. UNIX style compilers have historically used chars signed by default, but this behaviour may not be efficient on small microprocessors. If the operation of sign-extending a *char* to an *int* costs more machine instructions than clearing the extra bits, the default attribute for a plain char type will be *unsigned*. Otherwise, it will be *signed*. Note that it is possible to modify the default behaviour of the COSMIC compilers by using a specific option on the parser (**-u**).

There is another way to define an integer variable, by defining the range of possible values. An **enumeration** is an integer type defined by a list of identifiers. The compiler assigns an integer value to each of these identifiers, or uses a specific value declared with the identifier.

Each of these identifiers becomes a new constant for the compiler, and by examining the smallest and the largest values of the enumeration, the compiler will choose the smallest integer type large enough to hold all the values. The enumeration is a convenient way to define a set of codes and at the same time the most efficient variable type to hold all of these codes.

Bit Type

The bit type is

`_Bool`	1 bit

The bit type is an extension to the C standard and is implemented conformant to the boolean type of the C99 standard. It is not available on all the Cosmic compilers, and mainly for those target processors providing efficient instructions for bit handling.

Objects of this type are packed into bytes or words in memory and allocated to a memory section appropriate for access by efficient bit handling instructions. Such objects have only the two possible values true and false usually coded as 1 and 0.

Real Types

A real type is one of

`float`	4 bytes
`double`	8 bytes
`long double`	more than 8 bytes

The type **long double** is generally used to describe internal types of arithmetic coprocessors, handling reals on 9 or 10 bytes, in order to avoid any loss of precision when storing an intermediate result in memory. The physical coding of a real number is not fixed by the ANSI standard, but most compilers use the IEEE754 encoding mechanism. It is probably not the most efficient for a small microprocessor, but it is a convenient way to unify the encoding of reals over the various compilers and processors. With the IEEE754 coding, a *float* variable holds real numbers with 7 digits precision and an exponent range of 10 to the power ± 38. A *double* variable holds real numbers with 14 digits precision and an exponent range of 10 to the power ± 308.

For some small target processors, only the type **float** is supported by the compiler. In that case, all the types are allowed in the program syntax, but they are all mapped to the type *float* internally. This mechanism is also available as an option for larger targets if the application does not require a very accurate precision. It reduces the memory usage and the time needed to perform the operations.

Fractional Types

A fractional type is one of

`_Fract`	2 bytes
`long _Fract`	4 bytes

The fractional types are an extension of the standard types described in the standard ISO document DTR 18037. A fractional type encodes a real value in the range **[-1, 1[**, represented by a binary value equivalent to the integer value of the real number multiplied by 32768 for **_Fract** or 2147483648 for **long _Fract**. The type *_Fract* can also be written **short _Fract**. Fractional types are using the same operators than real types, with the addition of left and right shift operators. The fractional types are not supported for all the target processors.

Pointers

The C language allows more complex types than the simple numerical types. The first one is the **pointer** which allows for simple handling of addresses. A *pointer* is a variable, which contains an **address**. In order to know what to do with that address, a type is associated with the pointer.

A pointer takes the type of the object pointed at by the pointer value. A pointer in C is associated with the operator `*`, which is used to declare a pointer and to designate the object pointed at by the pointer. A pointer allows access to any location of the processor memory without any control other than any hardware control mechanism. It means that a pointer is as convenient as it is dangerous. In general, the C language does not verify anything at execution time, producing good efficiency, but with the same security as assembler, meaning none. All values are possible, but the C language reserves the value zero to designate an invalid pointer, although nothing stops a program accessing memory at address zero with a pointer. The size of a pointer in memory depends on the target processor. Most of the small microprocessors address 64K of memory and use 16 bit addresses. A pointer is then equivalent to an **unsigned short**. Some processors allow several memory spaces with various addressing mechanisms. To allow the best efficiency, the compiler supports three different pointer types by defining a size attribute along with the pointer:

a **`tiny`** pointer is a one byte address (*unsigned char*)
a **`near`** pointer is a two byte address (*unsigned short*)
a **`far`** pointer is a four byte address (*unsigned long*)

The compiler allows the user to select a **memory model** to choose a default size when no attribute is explicitly specified. In most of the cases, the *near* pointer will be used by default. When the addressing space is not large enough to hold the full application, a possible solution is to use a **bank switched** mechanism to enlarge the physical memory space, while keeping the same microprocessor and its logical memory addressing capabilities. In that case, *far* pointers are used to hold a two component address, consisting of a bank number and a logical address. Bank switching is mainly used for functions and sometimes allowed on data accesses depending on the target processor.

Arrays

The next complex type is common to several languages. An **array** may be defined as a collection of several objects of the same type. All these objects are allocated contiguously in memory and they may be accessed with an **index**, designating the rank of an object within an array.

An array is defined with the type of its elements and a dimension. An index always starts at zero in C. An index is applied to an array with the `[]` operators. No control is done to check that the index is in the proper range. It is possible to define an array of all the existing data types of C. An array in C has only one dimension. Multidimensional arrays are possible by creating arrays of arrays.

Structures

A **structure** may be defined as a collection of several objects of different types. No index can be used as objects may have a different size.

Each object will be accessed individually by its name. Such a structure member will be called a **field**. A structure is a convenient way of grouping together objects related to a specific feature. All the members are allocated in memory in the order defined by the structure and contiguously. Note that some processors have addressing restrictions and need sometimes to align data in memory on even boundaries for instance. The C compiler will respect these restrictions and will create holes if necessary to keep alignment. This means that a structure may not be allocated exactly in the same way depending on the target processor. Most small microprocessors do not require any alignment. The size of a structure in memory will be the sum of the size of each of its fields, plus if necessary the size of the padding holes. A structure may contain special fields called **bitfields** defining objects with a size smaller than a byte. Such objects are defined with a bit size.

Unions

A **union** is a variant of a structure. In a *structure*, all the members are individual objects allocated contiguously. In a *union*, all the members refer to the same object, and are allocated at the same address. All the members of a union are equivalent, and the size of a *union* will be the size of its largest member. A *union* is a convenient way to save memory space when a location may be used with different formats depending on the context.

Enumerations

An **enumeration** is an integer object defined by the list of its possible values. Each element is an integer constant and the compiler will choose the smallest integer type to implement such an object.

Functions

The C language defines a **function** as an object. This allows the same syntax to be used for a function declaration, and also allows a pointer to point at a function. A function is a piece of code performing a transformation. It receives information by its arguments, it transforms these input and the global information, and then produces a result, either by returning a value, or by updating some global variables. To help the function in its work, it is allowed to have local variables which are accessed only by that function and exist only when the function is active. Arguments and local variables will generally be allocated on the stack to allow recursivity. For some very small microprocessors, the stack is not easily addressable and the compiler will not use the processor stack. It will either simulate a stack in memory if the application needs to use recursivity, or will allocate these variables once for ever at link time, reducing the *entry/exit* time and code load of each function, but stopping any usage of recursivity.

When a function call is executed the arguments are copied onto the stack before the function is called. After the call, the return value is copied to its destination and the arguments are removed from the stack. An argument or a return value may be any of the C data objects. Objects are copied, with the exception of arrays, whose address is copied and not the full content. So passing an array as argument will only move a pointer, but passing a structure as argument will copy the full content of the structure on to the stack, whatever its size.

A C program may be seen as a collection of objects, each of these objects being a variable or a function. An application has to start with a specific function. The C environment defines the function called **main** as the first one to be called when an application is starting. This is only a convention and this may be changed by modifying the compiler environment.

CHAPTER
3

Declarations

An object **declaration** in C follows the global format:

<class> <type> <name> <initialization> ;

This global format is shared by all the C objects including functions. Each field differs depending on the object type.

<class> is the **storage class**, and gives information about how the object is allocated and where it is known and then accessible.

<type> gives the basic **type** of the object and is generally completed by the name information.

<name> gives a **name** to the object and is generally an identifier. It may be followed or preceded by additional information to declare complex objects.

<initialization> gives an **initial value** to the object. Depending on how the object is allocated, it may be static information to be built by the linker, or it may be some executable code to set up the variable when it is created. This field is optional and may be omitted.

Each declaration is terminated by the semicolon character `;`. To be more convenient, a declaration may contain several occurrences of the pair *<name> <initialization>*, separated by commas. Each variable shares the same *storage class* and the same basic *type*.

Integers

An **integer** variable is declared using the following basic types:

`char`
`short` or `short int`
`int`
`long` or `long int`
`long long` or `long long int`

These basic types may be prefixed by the keyword **signed** or **unsigned**. The type **unsigned int** may be shortened by writing only **unsigned**. Types *short*, *int*, *long* and *long long* are *signed* by default. Type *char* is defaulted to either signed *char* or *unsigned char* depending on the target processor. For most of the small microprocessors, a plain *char* is defaulted to **unsigned char**. The type **long long** is not supported by all the Cosmic compilers.

Reals

A **real** variable is declared using the following basic types:

`float`
`double`
`long double`

In most of the cases, type **long double** is equivalent to type **double**. For small microprocessors, all real types are mapped to type **float**.

Fractionals

A **fractional** variable is declared using the following basic types:

`_Fract` or `short _Fract`
`long _Fract`

The fractional types are not supported by all the Cosmic compilers.

For these numerical variables, an initialization is written by an equal sign **=** followed by a constant. Here are some examples:

```
char c;
short val = 1;
int a, b;
unsigned long l1 = 0, l2 = 3;
```

Bits

A **bit** variable is declared with the following basic type:

```
_Bool
```

These variables can be initialised like a numerical type, but the assignment rules do not match the integer rules as described in the next chapter. Here are some examples:

```
_Bool ready;
_Bool led = 0;
```

It is not possible to declare a pointer to a bit variable, nor an array of bit variables.

Pointers

A **pointer** is declared with two parameters. The first one indicates that the variable is a pointer, and the second is the type of the pointed object. In order to match the global syntax, the <*type*> field is used to declare the type of the pointed object. The fact that the variable is a pointer will be indicated by prefixing the variable name with the character *****. Beware that declaring a pointer does not allocate memory for the pointed object, but only for the pointer itself. The initialization field is written as for a numerical variable, but the constant is an address constant and not a numerical constant, except for the numerical value zero which is conventionally representing the **NULL** pointer. Here are some examples:

```
char *pc;       /* pointer to a char */
int *pv = &a;   /* initialized to address of a */
short *ps = 0; /* initialized to NULL pointer */
```

A pointer can be declared with the special type **void**. Such a pointer is handled by the compiler as a plain pointer regarding the assignment operations, but the object pointed at by the pointer cannot be accessed directly. Such a syntax is interesting when a pointer has to share different types.

Arrays

An **array** is declared with three parameters. The first one indicates that the variable is an array, the second indicates how many elements are in the array and the third is the type of one element. In order to match the global syntax, the <*type*> field is used to declare the type of one element. The fact that the variable is an array and its dimension will be indicated by adding the dimension written between square brackets `[10]` after the name. The dimension is an integer constant which may be omitted in some cases. An array initialization will be written by an equal sign **=** followed by a list of values placed between curly braces, and separated by commas. Here are some examples:

```
char tab[10];   /* array of 10 chars */
int values[3] = {1, 2, 3};
```

An initialization list may be smaller than the dimension, but never larger. If the list is smaller than the specified dimension, the missing elements are filled with zeroes. If an initialization list is specified, the dimension may be omitted. In that case, the compiler gives the array the same length than the specified list:

```
int values[] = {1,2,3};/* array of 3 elements */
```

An array of *char* elements can be initialized with a text string. written as a sequence of characters enclose by double quotes characters:

```
char string[10] = “hello”;
```

The missing characters are filled with zeroes. Because a text string is conventionally ended by a NULL character, the following declaration:

```
char string[] = “hello”;
```

defines an array of 6 characters, 5 for the word `hello` itself, plus one for the ending NULL which will be appended by the compiler. Note that if you write the following declaration:

```
char string[5] = “hello”;
```

the compiler will declare an array of 5 characters, and will not complain, or add any NULL character at the end. Any smaller dimension will cause an error.

All these declarations may be applied recursively to themselves, thus declaring pointers to pointers, array of arrays, arrays of pointers, pointers to arrays, and so on. To declare an array of 10 pointers to chars, we write:

```
char *ptab[10];
```

But if we need to declare a pointer to an array of 10 chars, we should write:

```
char *tabp[10];
```

Unfortunately, this is the same declaration for two distinct objects. The mechanism as described above is not enough to allow all the possible declarations without ambiguity. The C syntax for declaration uses priorities to avoid ambiguities, and parentheses to modify the order of priorities. The array indicators `[]` have a greater priority than the pointer indicator `*`. Using this priority, the above example will always declare an array of `10` pointers.

To declare a pointer to an array, parentheses have to be used to apply first the pointer indicator to the name:

```
char (*tabp)[10];
```

Modifiers

A declaration may be completed by using a modifier. A modifier is either of the keywords **const** and **volatile** or any of the **space** modifiers accepted by the compiler. A space modifier is written with an *at* sign `@` followed by an identifier. The compiler accepts some predefined space modifiers, available for all the target processors, plus several target specific space modifiers, available only for some target processors. The COSMIC compiler provides three basic space modifiers for data objects: **@tiny**, **@near** and **@far**. **@tiny** designates a memory space for which a one byte address is needed. **@near** designates a memory space for which a two byte address is needed. **@far** designates a memory space for which a four byte address is needed. The compilers are provided with one or several different memory models implementing various default behaviours, so if none of these space modifiers is specified, the selected memory model will enforce the proper one.

The **const** modifier means that the object to which it is applied is constant. The compiler will reject any attempt to modify directly its value by an assignment. A cross compiler goes further and may decide to locate such a *constant* variable in the code area, which is normally written in a PROM. A *const* object can be initialized only in its declaration. When the initialised value of a const variable is known to the compiler, because it is declared in the current file, an access to this variable will be replaced by direct use of the initialised value. This behaviour can be disabled by the **-pnc** compiler option if such an access must be implemented by a memory access.

The **volatile** modifier means that the value of the object to which it is applied may change *alone*, meaning without an explicit action of the program flow. This is the case with an input port, or with a variable updated by an interrupt function. The effect of such a directive is to stop the compiler optimizing the accesses to such a variable. In the following example:

```
char PORTA;

PORTA = 1; /*create a short pulse on bit 0 */
PORTA = 0;
```

The first assignment will be optimized out by the compiler as the `PORTA` variable is supposed to be a plain memory location which is not used between the two assignments. If such a variable is matching a physical output port, it must be declared as a *volatile* object:

```
volatile char PORTA;
```

A modifier applies to the current element being defined. When applied to a pointer, a modifier may affect the pointer itself, or the pointed object, depending on its position in the declaration. If the modifier is placed before the * character, it affects the pointed object. If the modifier is place after the * character, it affects the pointer.

```
const char *pc; /* pointer to a constant char */
pc = qc;        /* OK */
*pc = 0;        /* ERROR */
```

The first assignment modifies the pointer itself and is allowed. The second assignment tries to modify the pointed *const* object, and is then **not** allowed.

```
char * const pc;/* constant pointer to a char */
pc = qc;        /* ERROR */
*pc = 0;        /* OK */
```

The first assignment tries to modify a *const* pointer and is then **not** allowed. The second assignment modifies the pointed object and is allowed.

```
const char * const pc;
```

The **pc** object is declared as a *const* pointer to a *const* object, so no assignment to the pointer itself or to the pointed object will be allowed. Such an object probably needs to be initialized within the declaration to have any useful meaning for the program.

The compiler also implements special modifiers whose usage depends on the target processor:

The **@packed** modifier is used when the target processor requests an even alignment on word or larger objects, in order to stop the alignment for the specified object, assuming that unaligned accesses are still supported by the processor. It can also be used on a function definition to stop alignment on local variables thus shortening the local size.

The **@nostack** modifier is used to allocate a function stack frame (locals and arguments) in static memory instead of using the physical stack. This feature is interesting for small processors where the physical stack is not easily accessible. In such a case, the memory used for those local areas is allocated and optimized by the linker in order to consume the smallest amount of memory as possible.

The **@inline** modifier is used on a function declaration to let the compiler inlining the function wherever it is called. A C99 extension allows the **inline** keyword to be used as a storage class on a function declaration to activate the same feature. Once declared, an inline function will be inlined for any subsequent call found in the same file. If such a function must be inlined globally, its declaration should be placed in a header file included in all source files. An inline function cannot contain static locals and cannot call itself recursively.

Structures

A **structure** is declared by declaring all of its fields. They are grouped between curly braces and are preceded by the keyword **struct**. A structure, as a type, may be named to be reused later. This feature avoids repeating the full content of the structure. Such a name is called a **tag name** and is placed between the keyword *struct* and the opening curly brace.

```
struct {          /* unnamed structure */
     int a;       /* first field is an int */
     char b;      /* second is a char */
     long c;      /* third is a long */
     }
```

This set will fill the <*type*> field of the global declaration syntax. There is no modification of the <*name*> field.

```
struct node {     /* named structure */
     int value;
     struct node *left;
     struct node *right;
     } n1, n2;
```

declares two structures of type `node`. A reference to a not yet completed structure is possible as long as it is used to define pointers. Once a tag name has been associated to a structure, any further declaration does not need to repeat the structure content:

```
struct node n3, n4;
```

A structure initialization will be written by an equal sign = followed by a list of values placed between curly braces, and separated by commas. Each field is initialized by the corresponding value, converted to the corresponding type.

```
struct {
     char a;
     int b;
     long c;
     } st = {1, 2, 3};
```

field `a` is initialized with value 1
field `b` is initialized with value 2
field `c` is initialized with value 3

If the initialization list contains less values than structure fields, the missing fields are initialized to zero.

A **bitfield** is declared by suffixing the field name with a colon followed by the number of bits used by the bitfield. A bitfield can be declared in any **integer** type (the ANSI standard allows only types **int**, **unsigned int** and **signed int** for bitfields. The other possible types are extensions allowed by the Cosmic compiler, and by most compilers targeting microcontrollers). The reference type used is considered to be the *allocation unit*, meaning that an integer of that type is open and filled until there is no more space available. Any bitfield overlapping an allocation unit boundary is allocated in a new integer.

```
struct {
     char a:4;
     char b:3;
     char c:2;
     }
```

This structure defines 3 bitfields based on a *char* allocation unit. A first char is open and bitfields **a** and **b** are allocated inside. There is not enough space available to allocated bitfield **c**. A new char is then open and bitfield **c** is allocated inside. This structure is thus occupying 2 bytes in memory.

A bitfield without name will be used to skip the corresponding amount of bits in the allocation unit. A zero bitfield width is forcing the opening of a new allocation unit.

The ANSI standard does not define in which order bitfields are filled. The COSMIC compiler fills bitfields from the less significant bit to the most significant bit by default. This ordering can be reversed by using the **+rev** compiler option. The ANSI standard also limits the allocation unit type to *int*, *unsigned int* and *signed int*. The COSMIC compiler allows all the integer types as an extension.

By default, the compiler does not allocate the unused part of the last bitfield if its size is larger or equal to a byte. This process can be disabled by using the **-pnb** compiler option.

The ANSI allows bitfields to be signed or unsigned depending on their declaration, but allows the compiler to default basic types to unsigned for efficiency. The COSMIC compiler does not implement signed bitfields for efficiency reason. By default, all types will be considered as unsigned even if explicitly declared with the *signed* keyword. In order to avoid portability problems, this situation can be reported with an error message by using the **-pcsb** compiler option.

A C99 extension allows the last field of a structure to be an array declared without any explicit dimension. This field does not count in the structure size, but will allow whatever memory is found at that address to be accessed. This feature is described as *flexible array*. Typically, this construct is used in a *malloc* call to allocate a memory area large enough to contain the structure followed by an array with a variable size.

```
struct id {
     int length;
     char name[];
     };

     p = malloc(sizeof(struct id) + 20);
```

The memory area created by the *malloc* matches a structure, which would be defined like:

```
struct id {
     int length;
     char name[20];
     };
```

Unions

A **union** is declared like a structure, but the keyword *union* replaces the keyword *struct*. A *union* may be initialized, but as all the fields are located at the same address, it is seen as a single variable for the initialization, which will be done using the first field of the union.

```
union {
     char a;
     int b;
     long c;
     } u = {1};
```

field `a` is initialized with the value 1 on a `char`. It is then more convenient to define the largest field first to be sure to initialize all the union byte.

A *tag name* may be specified on a *union*. Tag names are in the same name space, so a union tag name cannot be the same as a structure tag name.

Enumerations

An **enumeration** is declared with a syntax close to the structure/union declaration. The list of fields is replaced by a list of identifiers. The keyword **enum** replaces the keyword *struct* or *union*. A tag name may also be specified, sharing the same name space than the structure and union tag names. Each identifier will be assigned a constant value by the compiler. An enumeration variable will be allocated as a *char*, a *short* or a *long* depending on the range of all the identifiers. This means that the compiler always needs to know all the enum members before it allocates an *enum* variable. If the **-pne** option has been set, an *enum* variable is always allocated as an *int* and then there is no need to know the *enum* members before to allocate the variable.

Note that the COSMIC compiler allows *long* enumerations as an extension to the ANSI standard.

```
enum {blue, white, red} flag;
```

The names `blue`, `white` and `red` are three new constants. Values are assigned to the names starting at zero, and incrementing by one for each new name. So `blue` is zero, `white` is one and `red` is two. The variable `flag` will be declared as a plain char as a byte is large enough to hold values from zero to two.

It is possible to define directly the value of a name by adding the character = followed by a value to the name.

```
enum {blue, white = 10, red} flag;
```

`blue` is still zero, `white` is now ten, and `red` is eleven, as the internal counter is incremented from the previous value. These names become new constants and may be used in any expression, even if they are not assigned to an enumeration variable. In the same way, an enumeration variable may be assigned with any kind of integer expression. An *enumeration* may be initialized as an integer variable.

Functions

A **function** is declared with three parameters. The first one indicates that the object is a function, the second gives the argument list, and the third is the type of the returned value. In order to match the global syntax, the <*type*> field is used to declare the type of the returned value. The fact that the object is a function will be indicated by adding the argument list written between parentheses `()` after the name. The <*type*> field is used to declare the type of the returned value. If the function has nothing to return, the <*type*> field is replaced by the keyword `void` meaning that nothing is returned from the function. This syntax will also allow the compiler to detect any invalid usage of the function, if it is used in an expression or an assignment.

The argument list may be specified in two different ways. The first one which is the oldest is known as the **Kernigan and Ritchie** (K&R) syntax. The second has been introduced by the standardization and is known as the **prototyped syntax**.

The *K&R* syntax specifies the argument list as a list of identifiers separated by commas between the parentheses. This list is immediately followed by the full declaration of the identifiers specified in the list. An undefined identifier will be defaulted to an *int* argument.

```
int max(a, b)
     int a;
     int b;
```

The *prototyped* syntax specifies the argument list as a list of individual declarations separated by commas between the parentheses.

```
int max(int a, int b)
```

The *prototyped* syntax offers extra features compared with the *K&R* syntax. When a function is called with parameters passing, the compiler will check that the number of arguments passed matches the number in the declaration, and that each argument is passed with the expected type. This means that the compiler will try to convert the actual argument into the expected type. If it is not possible, the compiler will produce an error message.

None of these checks are done when the function is declared with the *K&R* syntax. If a function has no arguments, there should be no difference between the two syntaxes. To force the compiler to check that no argument is passed, the keyword **`void`** will replace the argument list.

```
int func(void)
```

The *K&R* syntax allows a variable number of arguments to be passed. In order to keep this feature with the prototyped syntax, a special punctuator has been introduced to tell the compiler that other arguments may be specified, with unknown types. This is written by adding the sequence `...` as last argument.

```
int max(int a, int b, ...)
```

The compiler will check that there are at least two arguments, which will be converted to *int* if they have a compatible type. It will not complain if there are more than two arguments, and they will be passed without explicit conversion.

The *prototyped* syntax is preferred as it allows for more verification and thus more safety. It is possible to force the COSMIC compiler to check that a function has been properly prototyped by using the **-pp** or **+strict** options.

The initialization of a function is the **body** of the function, meaning the list of statements describing the function behaviour. These statements are grouped in a **block**, and placed between curly braces `{}`. There is no equal sign as for the other initializations, and the ending semicolon is not needed.

```
int max(int a, int b)
     {
     /* body of the function */
     }
```

A function may have local variables which will be declared at the beginning of the function block. In order to allow the recursivity, such local variables and the arguments have to be located in an area allocated dynamically. For most of the target microprocessors, this area is the stack. This means that these local variables are not allocated in the same way as the other variables.

Storage Class

It is possible in C to have a partial control over the way variables are allocated. This is done with the <*class*> field defining the **storage class**. This information will also control the **scope** of the variable, meaning the locations in the program where this variable is known and then accessible. The *storage class* is one of the following keyword:

```
extern
static
auto
register
```

extern means that the object is defined somewhere else. It should not be initialized, although such a practice is possible. The *extern* keyword is merely ignored in that case. The definition may be incomplete. An array may be defined with an unknown dimension, by just writing the bracket pair without dimension inside.

```
extern int tab[];
```

A function may be declared with only the type of its arguments and of its return value.

```
extern int max(int, int);
```

Note that for a function, the *extern* class is implied if no function body is specified. Note also that if a complete declaration is written, useless information (array dimension or argument names) is merely ignored. The COSMIC compiler may in fact use a dimension array to optimize the code needed to compute the address of an array element, so it may be useful to keep the dimension even on an extern array declaration.

static means that the object is not accessible by all parts of the program. If the object is declared outside a function, it is accessible only by the functions of the same file. It has a **file scope**. If an object with the same name is declared in another file, they will not describe the same memory location, and the compiler will not complain. If the object is declared inside a function, it is accessible only by that function, just like a local variable, but it is not allocated on the stack, meaning that the variable keeps its value over the function calls. It has **function scope**.

auto means that the object is allocated dynamically, and this implies that it is a local variable. This class cannot be applied to an object declared outside a function. This is also the default class for an object declared inside a function, so this keyword is most of the time omitted.

register means that the object is allocated if possible in a physical register of the processor. This class cannot be applied to an object declared outside a function. If the request cannot be satisfied, the class is ignored and the variable is defaulted to an *auto* variable.

A *register* variable is more efficient, but generally only a few variables can be mapped in registers. For most of the small microprocessors, all the internal registers are used by the compiler, and the *register* class is always ignored. The *register* class may be applied to an argument. In that case, the argument is still passed on the stack, but is copied into a register if possible when the function starts. A *register* object may be used as a plain object, except that it is not possible to take its address, even if it has been mapped in memory.

Typedef

The C language allows the definition of a new type as a combination of existing types. The global declaration syntax is used with the special keyword **typedef** used in place of the <*class*> field. The <*name*> field describes a new type equivalent to the type described by the declaration.

```
typedef int *PINT;
```

declares the identifier `PINT` to be a new type equivalent to type *int* *. It may be used now as a basic type of the C language.

```
PINT ptab[10];
```

declares the variable `ptab` as an array of `10` elements, each element is an object of type `PINT`, meaning a pointer to an *int*. This declaration is equivalent to:

```
int *ptab[10];
```

The *typedef* feature is a convenient way to redefine all the types used by an application in a coherent and more verbose way than using only the basic C types.

Variable Scope

Once declared, an object can be hidden locally by another declaration.

A global variable (declared outside a function) has an *application scope*, meaning that it can be accessed by any part of the application, assuming it is properly declared in each file. When using the **static** keyword on its declaration, a global variable has a *file scope*, meaning that it can be accessed only inside the file where it is declared. In any other file, the same declaration will not access the same object. It means that a static declaration at file level is hiding a global level variable declared with the same name in another file.

A local variable (declared inside a function) has a *function scope*, meaning that it can be accessed only inside that function, even if declared with the **static** keyword. A function level object will hide any object declared at file or global level with the same name.

Absolute Addressing

The COSMIC compiler allows an object to be declared along with its address when it is known at compile time (I/O registers). The address is specified just after the declaration and is replacing the initialization part. It is prefixed by the `@` character:

```
volatile char PORTB @0x10;
```

Such a declaration is in fact equivalent to an extern declaration so it is not possible to initialize such a variable. This can also be applied to a function declaration if such a function already exists at a known address in the application. This cannot be used to locate a function defined in the application at a predefined address.

A bit variable can also be declared at an absolute location by defining its byte address followed by a colon character and the bit position:

```
_Bool PB3 @0x10:3
```

or, if the variable **`PORTB`** has been previously declared as before:

```
_Bool PB3 @PORTB:3
```

CHAPTER
4

Expressions

An **expression** is a set of variables, constants and operators which are combined together to provide a result. C expressions are more extensive than in other languages because the notion of a **result** is included in operations such as assignments and function calls.

Some special operations such as array indexing and pointer arithmetic can be the result of expressions. This makes C expressions a very powerful feature of the language.

Variables

Variables are simply expressed by their name, which is a C identifier which should have been previously defined, otherwise the compiler does not know the type of that variable and does not know how to access it. The only exception is that you can call a function which has not been declared. The compiler will define that unknown function as an external function returning an **int**. If the function is declared later in the same file and the actual return type does not match, or if strict checking options are used (**-pp**, **+strict**), the compiler will complain.

Constants

Constants can be expressed in several formats. Integer constants may be expressed in decimal, octal, hexadecimal or character format.

A decimal number is written with numerical characters in the set `0123456789` and does not begin with the character `0`. For example:

`125`	is a valid decimal constant
`2A58`	is NOT a valid decimal constant
`012`	is NOT a decimal constant

An octal number is written with numerical characters in the set `01234567` and begins with the character `0`.

For example:

`0177`	is a valid octal constant
`0A00`	is NOT a valid octal constant
`377`	is NOT an octal constant

An hexadecimal number begins with the characters `0x` or `0X` followed by characters in the set `0123456789ABCDEFabcdef`. Letters `a` to `f` or `A` to `F` represent elements from `10` to `15`. Lower case and upper case letters have the same meaning in a hexadecimal constant.

`0x125`	is a valid hexadecimal constant
`0Xf0A`	is a valid hexadecimal constant
`0xZZZ`	is NOT a valid hexadecimal constant
`xABC`	is NOT a valid hexadecimal constant

A character constant is written as a sequence of printable characters enclosed by single quotes. Although this definition allows several characters to be entered, a character constant is usually limited to one character. The resulting value of such a constant is equal to the actual code of that character (ASCII in most of the cases, including COSMIC compilers, but may be EBCDIC or any other depending on the target environment). If more than one character is specified in a character constant, the result value is built as if the sequence was a number expressed in base 256, where each character is a **digit** whose value is between 0 and 255.

For example:

`'A'`	is equivalent to 0x41 or 65 (ASCII)
`'AB'`	is equivalent to 0x4142 (not very useful)

Non printable characters may be entered with a special sequence called **escape sequence** and beginning with the backslash `\` character. Such a character may be expressed by its numerical code written in octal or in hexadecimal. There is no way to express it in decimal.

An octal value will be entered by up to three octal digits.

A hexadecimal value will be entered by the character `x` followed by hexadecimal digits (upper or lower case, including the `x` character). For example:

`\11`	is equivalent to 011
`\xFF`	is equivalent to 0xFF

A few control characters can be entered without using their numerical value. It is useful not only because you do not need to know their coding, but also because this gives you the ability to write portable code, as the result will depend on the target environment. A backspace character may have a different code if the code used is ASCII or EBCDIC.

Note that hexadecimal character constants are not limited to three digits like the octal constants. This may be a problem when used in string constants as shown below.

The control characters mapped (with their ASCII code) are the following:

`\a`	BELL	(`0x07`)
`\b`	BACKSPACE	(`0x08`)
`\t`	TAB	(`0x09`)
`\n`	LINE FEED	(`0x0A`)
`\v`	VERTICAL TAB	(`0x0B`)
`\f`	FORM FEED	(`0x0C`)
`\r`	CARRIAGE RETURN	(`0x0D`)

The C syntax allows a character constant to be immediately preceded by the **L** letter:

```
L'A'
```

Such a constant is a **wide character** whose coding may exceed the capability of a char object. This is mainly used for Japanese character sets. The COSMIC compiler accepts this syntax but ignores the prefix and uses the same encoding as plain character constants.

All these integer constants have at least the type **int**. If the entered constant exceeds the resolution of an *int* (which is signed), the final type will be the smallest one able to represent the constant. Note that for a decimal constant, if it cannot be expressed by an *int*, but by an **unsigned int**, the result will be of type **long** and not **unsigned int**, as a decimal constant usually expresses signed values. If the **long long** type is supported, when a decimal constant cannot be expressed by a *long*, the result will be of type **long long** and not **unsigned long**.

It is possible to modify the default type by using suffix characters. Suffix `l` or `L` forces the constant type to be **long**. If the **long long** type is supported, suffix `ll` or `LL` force the constant type to be **long long**. Suffix `u` or `U` forces the constant type to be *unsigned*. Both `L` and `U` suffixes, or `LL` and `U` suffixes, may be specified, but each suffix can be specified once only. The suffix has to follow the constant immediately without any white space in between. These suffixes allow portable code to be written avoiding different behaviour depending on the *int* type resolution.

For example, assuming that type *int* is 16 bits:

`125`	is type *int*
`125U`	is type *unsigned int*
`125L`	is type *long*
`125UL`	is type *unsigned long*
`0xffff`	is type *unsigned int* (*int* if 32 bits wide)
`65535`	is type *long* (*int* if 32 bits wide)

If the *long long* type is supported:

`1LL`	is type *long long*
`-1ULL`	is type *unsigned long long*

Floating point constants are entered with commonly used exponential notation. A floating point begins by a numerical digit, or the `.` character if the integer part is omitted. The exponent is written by the letter `e` or `E` followed by an exponent, which may be signed. The exponent may be omitted.

For example:

`1.5`	is a valid floating point constant
`1.23e3`	is a valid floating point constant
`.2e-3`	is a valid floating point constant
`3E+2`	is a valid floating point constant
`E-1`	is NOT a valid floating point constant

A floating point constant has the type **double** by default. The suffix character `f` or `F` forces the compiler to define the constant with the type **float**. The suffix `l` or `L` forces the constant to the type **long double**. Note that types *double* and *long double* are the same for most of the targets.

If fractional types are supported, the suffix `r` or `R` forces the constant to the type **_Fract**, and suffix `lr` or `LR` forces the constant to the type **long _Fract**.

For example:

`1.5`	is type *double*
`1.5F`	is type *float*
`1.5L`	is type *long double* (mapped to *double*)

If the fractional types are supported:

`0.15R`	is type *_Fract*
`0.15LR`	is type *long _Fract*

Strings

The C language also defines a **string** constant to ease character strings handling. Such a constant is written by a sequence of printable or non printable characters enclosed by double quote characters. Non printable characters are expressed the same way as individual character constants. Such a constant is built in memory, and its value is in fact the memory address of that location. A NULL character is appended to the sequence of characters to conventionally end the text string.

For example:

```
"hello"
```

is a pointer of type **char *** to a memory area containing the characters `h`, `e`, `l`, `l`, `o` and `\0`

Non printable characters may be entered as octal or hexadecimal constants:

`"\15\12"` or `"\x0d\0xa"`

define a string containing characters *carriage return* and *line feed*. These constants may interfere with the next character of the same string:

```
"\3123"
```

defines a string containing the two characters `'\312'` and `'3'`, and not the characters `'\3'` , `'1'`, `'2'`, `'3'`. Such a string can be entered by filling the octal constant up to three digits:

```
"\003123"
```

This trick cannot be used with hexadecimal constants because they have no length limit. In case of any conflict, the only solution is to use the octal syntax.

In order to allow long text strings to be specified, a string constant may be split in several contiguous substrings, with white spaces or line feeds in between. All the substrings will be concatenated by the compiler to obtain one long string which will be created in memory.

For example:

`"hello "  "world\n"` is equivalent to `"hello world\n"`

The C syntax allows a string constant to be immediately preceded by the **L** letter:

```
L"hello"
```

Such a constant is a **wide string** and each character is encoded as a **wide character**. This is mainly used for Japanese character sets. The COSMIC compiler accepts this syntax but ignores the prefix and uses the same encoding than plain character constants.

Sizeof

The C language also allows a special constant which is written:

```
sizeof expression
```

or

```
sizeof (type)
```

In both syntaxes, the result is an integer constant equal to the size in bytes of the specified object. If an expression is specified, the compiler evaluates it without producing any code. It just applies the conversion rules as explained below to get the resulting type of the expression. And then, as if a type was directly specified, the compiler replaces the full construct by a constant equal to the amount of bytes needed to store in memory an object of that type.

Operators

Operators are symbolic sequences which perform an operation from one or two operands (three for one operator), and provide a result. The C evaluation rules describe how the operands are prepared depending on their type, and what is the type of the result. First of all, operands are *promoted* to the **int** type. This means that if the type of any operand is smaller than the type *int*, it is converted to type *int*. If the type of an operand is larger or equal to type *int*, it is left unchanged. Then, if the operator requires two operands, and if those two operands do not have the same type, the operand with the smallest type is converted into the type of the largest. Then the operation is performed, and the result type is the same as the largest operand.

Converting an integer type into a larger one will keep the sign of the original operand if the smallest type is signed. Otherwise, the original value is zero extended to reach the size of the largest type. Converting any type to a larger floating point type will keep the sign if the original type is a floating point or a signed integer type, or will produce a positive value otherwise.

These rules imply that the result of any expression is at least an *int* value. If the application handles variables with types smaller than *int*, the code needed to evaluate expressions with these rules will contain a lot of implicit conversions which may appear to be useless. Hopefully, the compiler will shorten the code each time it detects that the final result is the same than if the full rules were applied.

Operators may have different behaviour depending on the way they are used and their operand types. This is a difficulty when learning C. Operands may be subexpressions. Any conflict between operators is resolved by priority rules, or by enclosing subexpressions in parentheses. For operators with the same priority, the grouping order allows grouping from left to right, or right to left. For most of the operators, the evaluation order of the operands for that operator is undefined, meaning that you cannot assume that left operand will be evaluated before the right operand. In most of the cases, it does not matter, but there are a few situations where the evaluation order is importance. In such a case, the expression should be split into several independent expressions.

Arithmetic operators

`a + b`	returns the addition of operands **`a`** and **`b`**
`+a`	returns the (promoted) value of operand **`a`**
`a - b`	returns the difference of operands **`a`** and **`b`**
`-a`	returns the negated value of operand **`a`**
`a * b`	returns the product of operands **`a`** and **`b`**
`a / b`	returns the division of operand **`a`** by operand **`b`**
`a % b`	returns the remainder of the division of operand **`a`** by operand **`b`**

All these operators apply to integer types. For these types, the division operator gives as result the integer quotient of the operands division and the remainder operator gives as result the integer remainder of the operands division. For signed operands, the remainder has always the same sign as the dividend.

All these operators except the remainder `%` apply to floating point types. Operators **`+`** and **`-`** also apply to pointers, but special rules are applied. The possible constructs are:

`p + i`	add an integer to a pointer (**`i + p`** is allowed)
`p - i`	subtract an integer from a pointer
`p - q`	subtract a pointer from another pointer

When adding or subtracting an integer to a pointer, the value of the promoted integer is multiplied by the size in bytes of the object pointed to by the pointer, before the addition or subtraction takes place. So adding one to a pointer modifies it so that it points to the next element rather than just the next byte.

For example:

```
short *p;
```

`p + 1`	actually adds the value 2 to the pointer value, because a **`short`** has a size of 2 bytes in memory.

The result of such an operation is a pointer with the same type as the pointer operand.

Subtracting a pointer from another first requires that both pointers have exactly the same type. Then, after having computed the difference between the two operands, the result is divided by the size in bytes of the related object. The result is then the number of elements which separate the two pointers. The result is always of type *int*.

Bitwise operators

`a & b`	returns the bitwise and of operands `a` and `b`
`a	b`	returns the bitwise or of operands `a` and `b`
`a ^ b`	returns the bitwise exclusive or of operands `a` and `b`
`a << b`	returns the value of promoted operand `a` left shifted by the number of bits specified by operand `b`
`a >> b`	returns the value of promoted operand `a` right shifted by the number of bits specified by operand `b`
`~a`	returns the one’s complement of promoted operand `a`

All these operators apply to integer types only. The right shift operator will perform arithmetic shifts if the promoted type of its left operand is signed (thus keeping the sign of the operand). Otherwise, it will perform logical shifts.

Boolean operators

Boolean operators create or handle logical values **true** and **false**. There is no special keyword for these values, numerical values are used instead. *False* is value zero, whatever type it is. *True* is any value which is not false, meaning any non zero value. This means that the result of any expression may be used directly as a logical result. When producing a logical true, the compiler always produces the value 1.

`a == b`	returns true if both operands are identical
`a != b`	returns true if both operands are different
`a < b`	returns true if operand `a` is strictly less than `b`
`a <= b`	returns true if operand `a` is less or equal to `b`
`a > b`	returns true if operand `a` is strictly greater than `b`
`a >= b`	returns true if operand `a` is greater or equal to `b`
`a && b`	returns true if operand `a` and operand `b` are true
`a || b`	returns true if operand `a` or operand `b` is true
`!a`	returns true if operand `a` is false

Both operators `&&` and `||` evaluate the left operand first. If the final result is reached at that point, the right operand is NOT evaluated. This is an important feature as it allows the second operand to rely on the result of the first operand.

Boolean operators can be combined together and such a syntax is accepted:

```
a < b < c
```

This is **not** behaving as if `b` was tested to be between `a` and `c`. The first compare (`a < b`) is evaluated and produces a logical result equal to 0 or 1. This result is then compared with `c`. Such a syntax is correct for the compiler which does not emit any error message, but it does not behave as you might expect.

When comparing any object to a constant, the compiler is checking if the constant does not exceed the maximum values possible for the object type. In such a case, the compiler is able to decide if the test is always true or always false and then optimizes the code by suppressing the useless parts. When the **+strict** option is used, the compiler outputs an error message when such an out of range compare is detected.

Assignment operators

Assignment operators modify memory or registers with the result of an expression. A memory location can be described by either its name, or an expression involving pointers or arrays, and is always placed on the left side of an assignment operator. Such an expression is called **L-value**. The expression placed on the right side of an assignment operator is then called **R-value**. Conversion rules differ from the ones used for other operators. The *R-value* is evaluated using the standard rules, and the compiler does not consider the *L-value* type. When the *R-value* has been evaluated, its resulting type is compared with the *L-value* type. If both are identical, the result is copied without alteration. If the *L-value* type is smaller than the *R-value* type, the result is converted into the *L-value* type, by truncating integers, or converting floating point types. If the *L-value* type is larger than the *R-value* type, the result is extended to the *L-value* type, by either sign extension or zero extension for integers, depending on the *R-value* type, or by converting floating point types. When the **+strict** option is used, the compiler outputs an error message when an assignment is truncating the *R-value*.

Assignment operators also return a value, which is equal to the *L-value* content after the assignment. Consider the result as if the *L-value* was read back after the assignment. In no case should the result be considered equal to the *R-value*, even if it is the case in some situations, for instance if both types are equal.

`a = b`	transfer `b` into `a`, and returns `a`

Assignments are possible between pointers and between structures. Both operands need to be of the same type. Note that for pointers, the type **pointer to void** is compatible with all the other pointer types. The integer constant **0** is also compatible with any pointer type, and is considered to be the **NULL** pointer. Pointers including modifiers are compatible if the left pointer has the same modifiers of the right pointer or more. However, this constraint is too restrictive for embedded applications, and the COSMIC compiler will simply ignore the modifiers when checking the pointers compatibility.

The COSMIC compiler allows pointers with different sizes, using the special modifiers **@tiny**, **@near** and **@far**. By default, the compiler will widen the size of a pointer, but will not narrow it, unless authorized by a parser option (**-np**).

Assignment to bit variables uses a different rule. The *R-value* is evaluated and compared to zero. The bit variable becomes true (1) if the *R-value* is different from zero, and false (0) if it is equal to zero. This differs from an assignment to a one-bit bitfield where the *R-value* is evaluated and truncated before beeing assigned to the bit destination.

Because the assignment operator returns a value, the following construct is possible in C:

```
a = b = c;
```

The behaviour of this expression is to evaluate `c` and to transfer the result into `b`, then read `b` and transfer it into `a`. Both sub-expression `a` and `b` must be L-values. Note that this expression is not identical to `a = c` and `b = c`, because if `a` and `b` have different types, the implicit conversions may alter the value transferred.

The C language allows several short-cuts in assignment expressions:

`a += b` is equivalent to `a = a + b`
`a -= b` is equivalent to `a = a - b`
`a *= b` is equivalent to `a = a * b`
`a /= b` is equivalent to `a = a / b`
`a %= b` is equivalent to `a = a % b`
`a &= b` is equivalent to `a = a & b`
`a |= b` is equivalent to `a = a | b`
`a ^= b` is equivalent to `a = a ^ b`
`a <<= b` is equivalent to `a = a << b`
`a >>= b` is equivalent to `a = a >> b`

These operators behave exactly as their equivalent constructs, with the same type limitations as the simple operators. In addition to the source code saving, the compiler knows that the L-value will be used twice, and uses this clue to produce the more efficient code, by avoiding computing the same expression twice.

The C language also defines two special assignment operators allowing pre or post incrementation or decrementation. These operators are applied to L-value expressions, and can be placed before or after to specify if the operation has to be done before or after using the L-value.

`++a`	equivalent to `a += 1`, returns `a` after increment
`a++`	equivalent to `a += 1`, returns `a` before increment
`--a`	equivalent to `a -= 1`, returns `a` after decrement
`a--`	equivalent to `a -= 1`, returns `a` before decrement

As for the previous operators, the type limitations are the same as for the simple **+** and **-** operators.

The resulting expression involving an assignment operator is no longer an L-value and cannot be re-assigned directly so the following expression

```
++ i --
```

is not valid in C, because if the `i` name is an L-value then `++i` cannot be an L-value so the `--` operator cannot be applied to it.

Those operators should not be used several times on the same variable in the same expression:

```
i++ - i++
```

will result **1** or **-1** depending on the evaluation order.

Addressing operators

The C language defines pointers, and a set of operators which can be applied to them, or which allows them to be built.

`*ptr`	returns the value pointed by the pointer `ptr`
`&var`	returns the address of variable `var`
`tab[i]`	returns an array element

The `*` operator applied to a pointer returns the value of the object pointed by the pointer. It cannot be applied to a pointer to a function, or to a pointer to void. The type of the result is the type provided in the pointer declaration.

The `&` operator returns the address of its operand. The operand can be a variable, or any legal L-value. It is not possible to compute the address of a variable declared with the *register* class, even if the compiler did not allocate it into a physical register, for portability reasons. The type of the result is a pointer to the type of the operand.

The `[]` operator is the **indexing** operator. It is applied to an array or a pointer, and contains an integer expression between the square brackets. Such an expression:

`tab[i]` is equivalent to `*(tab + i)`

where the `+` operator behaves as for a pointer plus an integer. This equivalence gives some interesting results:

`tab[0]` is equivalent to `*tab`
`&tab[0]` is equivalent to `&*tab`, or simply `tab`

This last equivalence shows that an array name is equivalent to a pointer to its first element. It also explains why the indexing operator can be applied to an array or to a pointer. The index expression is not checked against the boundary limits of an array. If the index goes beyond an array limit, the program will probably not behave well, but the compiler will not give any error or warning.

When used in a *sizeof* construct, an array name is not equivalent to a pointer and the sizeof result is equal to the actual array size, and not to the size of a pointer.

Because the `+` operator is commutative, `*(tab + i)` gives the same result as `*(i + tab)`, so `tab[i]` can be written `i[tab]`, even if such a construct is not very familiar to computer language users.

Function call operator

The **function call** is an operator in C which produces a result value computed from a user defined piece of code, and from a list of arguments. A function is called by writing its name followed by an expression list separated by commas, enclosed by parentheses:

```
function ( exp_1, exp_2, exp_3 )
```

Each expression is an argument evaluated and passed to the function according to the expected type if there is a prototype for that function. The arguments are passed from the right to the left, but this does not mean that they are always evaluated from the right to the left, although this is true for most compilers.

Each argument follows the standard evaluation rules and will be promoted if necessary. This means that a char variable will be passed as an int, and a float variable passed as a double if both types are supported. For small processors, this mechanism may consume too much time and stack space, so the COSMIC compiler allows this default mechanism to be stopped by an option (**+nowiden**). In such a case, any variable is passed in its basic type, although expressions will be passed with their expected promoted resulting type, unless it is cast. When the function is declared with a prototype, each argument is cast to the declared type before being passed. Note that the widening control option is not always available, depending on the target processor’s ability to stack single bytes or not.

Once evaluated, arguments are usually copied onto a stack, either by a push or a store instruction. It may be a physical stack, or a piece of memory simulating a stack, or for small microcontrollers an piece of memory dedicated to the function. In any case, an argument is passed by copying its value somewhere in memory. Structures are copied completely while arrays and functions are replaced by their address. If a function alters an argument declared with a basic type, the copy only is modified. If an argument is declared with an array type, and if an array element is modified by an assignment, the original array is modified. If a function must modify the content of a variable, it is necessary to pass explicitly its address by using the 'address of' operator (**&**). Such an argument needs to be declared as a pointer to the type of the expected variable.

The result of such a function call is the value returned by the function with the expected type.

The K&R syntax does not require that a function is defined before it is called. The return type is assumed to be *int*. Using the ANSI prototypes or the strict checking options avoid most of the errors created by that kind of situation.

A function can be called either directly, or from a pointer variable declared as a pointer to a function. Assuming such a pointer is declared as follows (the parentheses are necessary to solve the priority conflict between the pointer indicator and the function call indicator):

```
void (*ptrfunc)(int, int);
```

the function call can be written using the expanded syntax:

```
(*ptr_func)(arg1, arg2);
```

or by a shortcut using the pointer as if it was a function name:

```
ptr_func(arg1, arg2);
```

Both syntaxes are valid and produce the same code.

The COSMIC compiler implements a special function call to produce inline assembly code interfaced with C objects. The **_asm** function receives a text string as a first argument containing the assembly source code. This code will be produced instead of calling an actual function. Extra arguments will behave as plain function arguments, and the compiler will expect to find any returned value in the expected location (depending on the type and on the target processor).

Conditional operator

The **conditional** operator is equivalent to an **if ... else** sequence applied to an expression. The expression:

```
test_exp ? true_exp : false_exp
```

is evaluated by computing the `test_exp` expression. If it is true, meaning not zero, then the `true_exp` expression is evaluated and the resulting value is the final result of the conditional expression. Otherwise, the `false_exp` is evaluated and the resulting value is the final result of the conditional expression. These two expressions must have a compatible type and the type of the final expression follows the standard rules.

Sequence operator

The **sequence** operator allows a single expression to be expressed as a list of several expressions. The expression:

```
exp_1 , exp_2
```

is evaluated by computing the first expression, and then by computing the second one. The final result of such an expression is the result of the second one. The result of the first one is simply trashed. It means practically that such a construct will be interesting only if the first expression *does* something, such as an assignment, a function call, an increment or decrement operation. This operator is not allowed directly in a function call sequence, because the comma is used as an argument separator. In such a case, the sequence operator can be used only within expression parentheses.

Conversion operator

The **cast** operator allows the result of an expression to be converted into a different type. The final type enclosed by parentheses, is prepended to the expression to be converted:

```
( new_type ) expression
```

The type specified between parentheses is called an *abstract type* and is written simply as a variable declaration where the object name is omitted. So basic types are simply written with their type name:

`(char) 1`	convert the constant 1 (whose default type is *int*) into a constant 1 with char type.
`(long) var`	convert the variable `var` whatever its type is to type *long*.

The first operation does not produce any code, as it is just an internal type change in the compiler. The second one will produce actual code, unless `var` already has type long. Note that a conversion between signed and unsigned objects of the same type does not produce any code. The cast result will behave with its new type.

The cast operator can be used to allow different pointers to be accepted as compatible:

```
char *pc;
int *pi;

pi = (int *)pc;
```

When using the **+strict** option, the **cast** operator will be used to force the compiler to accept a truncating assignment without producing any error message:

```
char c;
int i;

c = (char)i;
```

The **cast** operator may be used to override the default evaluation rules. Assuming both variables `a` and `b` are declared as **unsigned char**, the following expression:

```
a == ~b
```

may not behave exactly as expected. The promotion rules force `b` to be converted to an int before it is inverted, so if `b` is originally equal to `0xff`, it first becomes an **int** equal to `0x00ff` (unsigned promotion) and the inverted result is then `0xff00`. This value is compare with the promoted value of `a` and if `a` was originally `0x00`, the compare operator returns a false result as it is comparing `0x0000` with `0xff00`. In order to force the compiler to evaluate the complement operator on a char type, the expression has to be written:

```
a == (unsigned char)~b
```

In such a case, the promoted result of the complement operator `0xff00` is truncated to a char and the equality operator is comparing `0x00` with `0x00` and returns a true result.

Priorities

All these operators may be combined together in complex expressions. The order of evaluation depends on each operator priority. The expression:

```
a + b * c
```

will be evaluated as:

```
a + (b * c)
```

because in C (as in most of the computer languages), the multiplication `*` has a higher priority than the addition `+`. Some operators have the same priority, and in such a case, it is necessary to define in which order they are evaluated. This is called grouping order and may be left to right, or right to left. A left to right grouping is applied to the usual arithmetic operators, meaning that:

```
a + b + c
```

is

```
((a + b) + c)
```

A right to left grouping is applied to the assignment operators, meaning that:

```
a = b = c
```

is

```
(a = (b = c))
```

The C language defines 15 levels of priority, described here from the highest to the lowest, with their grouping order:

1	**Left to Right**	

	post increment/decrement
	array subscript
	function call
	structure/union member
	pointer to a member
		`i++ i--`
		`tab[i]`
		`func()`
		`str.a`
		`p->a`

2	**Right to Left**	

	sizeof operator
	pre increment/decrement
	address of
	content of
	unary plus/minus
	binary/logical not
	cast
		`sizeof i`
		`++i --i`
		`&i`
		`*p`
		`+i -i`
		`~i !i`
		`(type)i`

3	**Left to Right**	

	multiply
	divide
	remainder
		`i * j`
		`i / j`
		`i % j`

4	**Left to Right**	
	Addition substract	`i + j` `i - j`
5	**Left to Right**	
	left shift right shift	`i << j` `i >> j`
6	**Left to Right**	

	less than
	less than or equal
	greater than
	greater than or equal
		`i < j`
		`i <= j`
		`i > j`
		`i >= j`

7	**Left to Right**	

	equal to not equal	`i == j` `i != j`
8	**Left to Right**	
	binary and	`i & j`
9	**Left to Right**	
	binary exclusive or	`i ^ j`
10	**Left to Right**	
	binary or	`i | j`
11	**Left to Right**	
	logical and	`i && j`
12	**Left to Right**	
	logical or	`i || j`
13	**Right to Left**	
	conditional expression	`i ? j : k`
14	**Right to Left**	

	assignment
	multiply assign
	divide assign
	remainder assign
	plus assign
	minus assign
	left shift assign
	right shift assign
	and assign
	exclusive or assign
	or assign
		`i = j`
		`i *= j`
		`i /= j`
		`i %= j`
		`i += j`
		`i -= j`
		`i <<= j`
		`i >>= j`
		`i &= j`
		`i ^= j`
		`i |= j`

15	**Left to Right**	
	comma	`i, j`

There are a few remarks about these levels. The shift operators have a lower priority than the additive operators, although they are closer to multiplication/division operations. This may produce an unexpected result in the following expression:

```
word = high << 8 + low ;
```

Assuming that **`word`** is a short integer, **`high`** and **`low`** two unsigned chars. This expression, which is supposed to combine two bytes concatenated into a word, will not produce the expected result. This is because the addition has a higher priority than the left shift. The grouping is actually:

```
word = high << (8 + low) ;
```

This is clearly wrong. The result will be correct if the binary or operator is used:

```
word = high << 8 | low ;
```

or if the shift operation is enclosed by parentheses:

```
word = (high << 8) + low ;
```

Parentheses should be used to avoid any ambiguity, without overloading the expression and making it difficult to be read.

CHAPTER
5

Statements

Statements are language instructions which can be entered only inside a function body. They describe the function behaviour and they are placed in a function declaration:

```
return_type function_name(argument_list)
        {
        local_declarations
        statement_list
        }
```

A list of statements comprises several statements placed one after the other, without any separator. The C language uses a terminator character `;` to mark the end of a statement if it is necessary to avoid any ambiguity in the understanding. A statement may or may not use a semicolon as terminator, but two statements are never *separated* by semicolons, even if it looks like that when reading C code.

The sequence entered between the two curly braces is called a *block*, and a block is a statement. This will allow several statements to be assembled together, and to behave syntactically as a single statement, so in the following description, any indication of **statement** may be replaced by any of the C statements, including a block. The statements defined by the C language are as follows:

Block statement

The syntax of a block statement is:

```
{
declaration_list

statement_list
}
```

The declaration part is a list of standard declarations mainly used to declare local variables. The **register** class may be used here to get more efficient code. These local variables are created when the block is entered, and destroyed when the block is exited, so two contiguous blocks will overlay their local variables in the same area (the stack in most of the cases). Most of the compilers will compute the maximum size needed by all the embedded blocks and will create the locals frame once at the function entry. The overlapping will be implemented by using same offsets for overlapping variables.

The statement part is a list of C statements as described below.

Expression statement

The syntax of an expression statement is:

```
expression;
```

using an **expression** as described before, and terminated be a **semicolon**. Any expression can be used, even if it does not perform any physical operation, but most of those expressions will be assignments, function calls or increment/decrement operations. Note that it is possible to omit the expression, thus leaving a semicolon alone. This is the empty statement, equivalent to a **nop**, but it is syntactically a valid statement.

```
a = b + 1;
func(1, x);
++y;
```

are operative expressions, as they modify the program status by changing the value of a memory cell or a register, or by calling a function which is supposed to do something.

A useless expression such as:

```
a;
```

is permitted by the syntax, and does not produce any code in most of the cases. This is used when the variable is declared with the **volatile** attribute to force the compiler to produce a **load** instruction. The variable is just read, which is important for some peripheral registers which need to be read in order to clear interrupt flags for instance.

An empty statement will be simply:

```
;
```

and will be used each time a statement has to be entered, but where nothing has to be done.

Note that when used alone, the `++` and `--` operators have the same behaviour on both sides of the modified objects:

```
x++;
++x;
```

are equivalent and produce the same code.

If statement

The syntax of an **if** statement is:

```
if ( expression )
        statement
```

or

```
if ( expression )
        statement
else
        statement
```

In these two syntaxes, the parentheses around the expression are part of the syntax, and not subexpression parentheses, so they need to be entered. Note that there is no **then** keyword, as the closing brace behaves exactly the same way. The behaviour of such a statement is as follows. The expression is first evaluated, and the result is checked against the two possible cases: true or false. If the expression is a comparison, or a set of combined comparisons with logical and/or operators `&&`, `||`, the result is the result of the comparison evaluation. If the expression does not use any comparison, the result has to be a numerical value or an address (from a pointer). The result is true if the value is not zero, and false is the result is zero. This is equivalent to an implied comparison with zero:

`if (a + b)` is equivalent to `if ((a + b) != 0)`

In the first syntax, the following statement is executed only if the result of the expression is true. Otherwise it is skipped.

```
if (a > b)
     b = a;
```

In this example, variables `a` and `b` are compared, and if `a` is greater than `b`, then `b` is set to the value of `a`.

In the second syntax, the following statement is executed only if the result of the expression is true, and in this case, the statement following the **else** keyword is skipped. Otherwise, the following statement is skipped, and the statement following the *else* keyword is executed.

```
if (a < 10)
     ++a;
else
     a = 0;
```

In this example, variable `a` is compared to the constant `10`. If a is smaller than `10`, it is incremented. Otherwise, `a` is reset to zero thus implementing a looping counter from zero to ten included.

If/else statements may be embedded. In such a case, an *else* statement always completes the closest *if* statement, unless properly enclosed with block braces. In this an example:

```
if (a < 10)
        if (a > 5)
                b = 1;
else
        b = 0;
```

the text formatting has no influence on the compiler, and the program will behave as if it were written:

```
if (a < 10)
        {
        if (a > 5)
                b = 1;
        else
                b = 0;
        }
```

because the *else* statement is associated with the closest *if* statement. To achieve the behaviour suggested by the text formatting, the program should be written:

```
if (a < 10)
        {
        if (a > 5)
                b = 1;
        }
else
        b = 0;
```

Now, the second *if* is part of a full block which becomes the first statement of the first *if*. The *else* statement can only be associated with the first *if* statement.

While statement

The syntax of a **while** statement is:

```
while ( expression )
        statement
```

This statement is used to implement a loop controlled by an expression. If the result of the expression is true, the following statement is executed, and the expression is re-evaluated to decide if the iteration can continue. When the expression is false, the following statement is skipped, and the loop is exited. As the expression is evaluated first, the loop statement will be executed zero or more times.

```
while (p < q)
      *p++ = '\0';
```

In this example, assuming that **p** and **q** are two pointers to char variables, the loop will clear all the characters between pointers **p** and **q** if at the beginning, **p** is smaller than **q**. This can be expanded in such a way:

```
while (p < q)
      {
      *p = '\0';
      ++p;
      }
```

if the increment operator is not used combined with the indirection. This time we have two different statements which need to be placed into a block statement for both to be executed both while the condition is true.

The empty statement can be used to implement a wait loop:

```
while (!(SCISR & READY))
      ;
```

Do statement

The syntax of a **do** statement is:

```
do
      statement
while ( expression ) ;
```

This statement is also used to implement a loop controlled by an expression, but here the statement is executed first, and then the expression is evaluated to decide if we loop again or if we stop there. This statement is terminated by a semicolon. The loop statement will be executed one or more times.

```
do
        ok = do_it();
while (!ok);
```

In this example, the function `do_it()` is executed until its return value is not zero.

For statement

The syntax of a **for** statement is:

```
for (expression_1 ; expression_2 ; expression_3)
        statement
```

The **for** statement is also used to implement a loop, but in a more powerful way than the previous ones. This instruction is equivalent to the following construct:

```
expression_1 ;
while ( expression_2 )
        {
        statement
        expression_3 ;
        }
```

The first expression is the loop initialization, the second expression controls the loop iteration, and the third expression passes to the next iteration. The statement is the *loop* body. This syntax, although functionally equivalent, has the advantage to include in one instruction all the elements describing the loop:

```
for (i = 0; i < 10; ++i)
        tab[i] = 0;
```

In this example, the loop initialization (`i=0`), the loop control (`i < 10`) and the next element control (`++i`) are displayed on the same line and the code reading and understanding is enhanced. More complex controls may be implemented using this syntax, such as list walking:

```
for (p = list_head; p; p = p->next)
     p->value = 0;
```

which walks though a linked list and resets a field, assuming for instance the following declarations:

```
struct cell {
     struct cell *next;
     int value;
     } *p, *list_head;
```

The **for** statement allows some variations. Any of the three expressions may be omitted. The behaviour is simple for the first and the third expressions. If they are omitted, they do not produce any code. This is different for the second expression because it controls the loop iteration. If the second expression is omitted, it is replaced by an always true condition, meaning that this creates an endless loop.

An embedded program which never returns can then be started by:

```
main()
     {
     for (;;)
          operate();
     }
```

The function operate will be repeated infinitely. This is sometimes written:

```
     while (1)
          operate();
```

which produces absolutely the same result, as `1` is always not zero, meaning true. The first syntax just looks more aesthetic.

The sequence operator is useful when a **for** loop uses several control variables:

```
for (i = 0, j = 10; i < j; ++i, --j)
     x = tab[i], tab[j] = tab[i], tab[i] = x;
```

Because the C syntax allows infinite loops, it also provides instructions to exit such loops.

Break statement

The syntax of a **break** statement is simply:

```
break ;
```

This statement has to be placed inside the **body** statement (a block usually) of a *while*, *do* or *for* instruction. It stops the execution of the body statement and jumps to the end of the statement, behaving as if the controlling expression was giving a false result. The remaining instructions of the including block are simply skipped. The *break* statement is usually associated with an *if* statement to decide if the loop has to be exited or not.

```
while (p < q)
        {
        if (!*p)
                break;
        *p++ = 'A';
        }
```

In this example, the loop body sets a buffer to the character '`A`' while the `p` pointer is smaller than the `q` pointer. In the body statement, the *break* instruction is executed if the current character is a zero. This will exit the loop and the execution will continue from the statement following the while block. The *break* statement in such a case can be considered as a **and** condition combined with the while test, as the previous code could have be written:

```
while (p < q && *p)
        *p++ = 'A';
```

The **not** operator `!` has been removed as the *while* condition is a continuation test, and not a termination test.

When applied to a *for* loop, the *break* statement exits the equivalent body statement from the expanded while construct, meaning that the third expression, if any was specified, is not evaluated.

When several loop statements are embedded together, a *break* statement will be applied to the closest loop statement:

```
for (i = 0; i < 10; ++i)
     {
     while (valid(i))
          {
          if (tab[i] < 0)
               break;
          --tab[i];
          }
     tab[i] = 0;
     }
```

In this example, the *break* instruction will stop the while loop only, thus continuing the execution with the statement following the while block (`tab[i] = 0`).

Continue statement

The syntax of a **continue** statement is simply:

```
continue ;
```

This statement has to be placed inside the body statement of a *while*, *do* or *for* loop. Its behaviour is to abort the current iteration and to start a new one. Practically, it means that the program execution continues by re-evaluating the controlling expression. When applied to a *for* loop, the third expression, if any specified, is evaluated before evaluating the second expression.

```
for (i = 0; i < 100; ++i)
     {
     if (tab[i] == 10)
          continue;
     ++tab[i];
     }
```

In this example, a *for* loop is used to increment all the elements of an array up to a maximum value of `10`. The *continue* statement is executed if an element has already reached the value `10`. In this case, the `++i` expression is executed, thus skipping to the next element, before re-evaluating the test `i < 100` and continue the loop.

This statement is useful to avoid a deep embedding of blocks when dealing with complex control expressions inside a loop.

When used in embedded loop statements, a *continue* statement is applied to the closest loop statement, as for the *break* instruction.

Switch statement

The syntax of a **switch** statement is:

```
switch ( expression )
        {
case constant_1:
        statement_list
case constant_2:
        statement_list
default:
        statement_list
        }
```

A *switch* statement is followed by an integer expression and by a block containing special **case** labels defining some entry points. A *case* label is followed by a constant expression, defining an integer value known at compile time. A *case* label defines only one value, but several different values may be associated together in defining several case labels without any statement between them. All the values entered in *case* labels must be distinct from the others. The optional **default** label has no parameter.

The behaviour of such a statement is as follows. The expression is evaluated and gives a numerical result. The program will then search for a case label defined with a value equal to the expression result. If such a label is found, the execution continues from the next statement following the label, until the end of the block statement. Execution will not stop when crossing any other case or default label. This behaviour is not common to the other high level languages and needs to be clearly stated. It is possible to stop executing the block by entering a break statement which will then make the switch behaviour look like the other languages. In fact, the *switch* statement can be compared to a computed jump, or computed goto in some languages (basic, fortran).

If no *case* label matches are found, there are two possible behaviours. If a *default* label has been defined, the execution continues from the next statement following it. Otherwise, the full block is skipped.

```
switch ( get_command() )
     {
case 'L':
     load();
     break;
case 'E':
     edit();
     break;
case 'X':
     save();
case 'Q':
     quit();
     break;
default:
     bark();
     break;
     }
```

In this example, a *switch* statement is used to decide what to do from a command letter. Each case statement activates one function and is followed by a break to avoid executing the other statements.

The case **X** shows a possible usage of the linear feature of a *switch* by executing the *save()* function, then continuing by the following *quit()* function, implementing the **Q** command as a direct exit, and the **X** command as a save and exit. The *break* statement following the last *case* or *default* label is basically useless. It is nevertheless good to have it to avoid forgetting it if the *switch* is extended later... Note that if all the labels are associated with a *break* statement, the display order has no importance, including the default label which can be placed anywhere.

Although the *break* statement has a meaning inside a switch statement, the *continue* statement has no effect, and will be refused as not being inside a loop statement. This may happen if the *switch* statement is itself inside a loop statement:

```
for (i = 0; i < 100; ++i)
        {
        switch (tab[i])
                {
        case 0:
                led = 1;
                break;
        case 1:
                led = 0;
                continue;
                }
        ++tab[i];
        }
```

In this example, the *break* statement will exit the *switch* statement, and continue execution at statement `++tab[i]`, while the *continue* statement will be applied to the *for* statement, as it has no meaning for the *switch* statement. The execution will continue at the `++i` of the *for* loop. In this case, it is not possible to use a *break* statement inside the *switch* block to exit the *for* loop. This can be achieved simply only with a *goto* statement, as explained below.

Goto statement

The syntax of a **goto** label is:

```
goto label ;
```

where `label` is a C identifier associated to a statement by the syntax:

```
label : statement
```

Despite all the high level language recommendations, a *goto* statement can be used wherever it saves extra code or variables, without breaking too much of the program's readability. *Goto*'s may also decrease the compiler optimization as it is more difficult to build a simple execution path when too many *goto*'s are involved.

A *label* is always followed by a statement, so it is not possible to jump to the end of a block by such a syntax:

```
          {
          ...
          if (test)
               goto exit;
          ...
     exit:
          }
```

Here, the label **`exit`** is followed by the closing curly brace, and this is a syntax error. This has to be written by using the empty statement:

```
     exit:
          ;
          }
```

Return statement

The syntax of a **return** statement is:

```
     return expression ;
```

or

```
     return ;
```

The *return* statement is used to leave a function, and to return to the expression which was calling that function. The first syntax is used to return a value from the function. The expression is evaluated, converted into the return type if necessary, and then placed in the return area (a conventional register or predefined memory location) associated with the called function. The second syntax is used when a function has nothing to return, meaning that it should have been declared as a **void** function. A *return* statement can be placed anywhere, usually associated with an *if* or a *switch* statement if it is not the last statement of a block or a function. Note that if a function does not contain a *return* statement at the end of the function block, the compiler automatically inserts one to avoid the program continuing with the next function, which is not very meaningful. This implicit *return* statement does not return any value. When the strict option is used (**+strict**), the compiler also checks that a function which has a return type is actually returning something.

There are no other statements in C. All the other features you can find in some other languages (input/output, file control, mathematics, text strings) are implemented by library routines. The C standard has also normalized the basic libraries, thus guaranteeing that those features can be used regardless of the compiler origin.

CHAPTER
6

Preprocessor

The C preprocessor is a text processor which operates on the C source before it is actually parsed by the compiler. It provides macro and conditional features very closed to the ones available with most of the existing assemblers.

The preprocessor modifies the C program source according to special directives found in the program itself. Preprocessor directives start with a sharp sign ‘#’ when found as the first significant character of a line. Preprocessor directives are line based, and all the text of a directive must be placed on a single logical line. Several physical lines can be used if all of them but the last one end with the continuation character backslash ‘\’.

There are three basic kinds of directives: macro directives, conditional directives and control directives.

The macro directives allow text sequences to be replaced by some other text sequences, depending on possible parameters.

The conditional directives allow selective compiling of the code depending on conditions most of the time based on symbols defined by some macro directives.

The control directives allow passing of information to the compiler in order to configure or modify its behaviour.

Macro Directives

The three macro directives are:

```
#define IDENT rest_of_the_line

#define IDENT(parameter_list) rest_of_the_line

#undef IDENT
```

The two first syntaxes allow a macro to be defined, and the third syntax allows a previous definition to be cancelled.

`IDENT` is a word following the rules for a C identifier, and may use lowercase or uppercase characters. For readability reasons, most macro names are entered uppercase only.

`rest_of_the_line` represents all the characters from the first significant character immediately following `IDENT` (or the closing brace for the second syntax) up to the last character of the line. This character sequence will then replace the word `IDENT` each time it is found in the C source after the definition.

The **#undef** directive will cancel the previous definition of a macro. No error is reported if the #undef directive tries to cancel a macro which has not been defined previously. However, you cannot redefine a macro which has already be defined. In such a case, it has to be first cancelled by a *#undef* directive before it is redefined.

The second syntax allows a replacement with parameters. Note that the opening brace has to follow immediately the last character of the macro name, without any whitespace. Otherwise, it is interpreted as the first syntax and the parameter list along with the parentheses will be part of the replacement sequence. Each parameter is an identifier, separated from the others by a comma.

```
#define SUM(a, b)   a + b
```

This macro defines the word `SUM` along with two parameters called `a` and `b`. Parameters should appear in the replacement part, and the macro should be used in the remaining C source with a matching number of arguments.

An argument will replace any occurrence of its matching parameter in the replacement list, before replacing the macro name with its arguments and the parentheses in the C source. If the program contains the following sequence:

```
x = SUM(y, z);
```

the final result will be:

```
x = y + z;
```

The preprocessor recognized **`SUM`** as a valid macro invocation, and successfully matched **`a`** with **`y`**, and **`b`** with **`z`**. The macro name and the arguments with the parentheses have been replaced by the replacement con tent of the macro **`a + b`** where **`a`** and **`b`** were replaced by their values **`x`** and **`y`**.

Arguments are also simple text strings separated by commas, so if an argument has to contain a comma, the full argument has to be enclosed with extra parentheses.

The preprocessor also allows two special operators in the replacement list of a macro with parameters.

The operator **#** placed before a parameter name will turn it into a text string by enclosing it by double quotes:

```
#define STRING(str)   # str
```

will transform:

```
ptr = STRING(hello);
```

into

```
ptr = “hello”;
```

This feature is interesting to use in conjunction with the string concatenation.

The operator ## placed between two words of the replacement sequence will concatenate them into a single one. A word may be a parameter but in such a case, the parameter will not be expanded before beeing concatenated.

```
#define BIT(var, bit)   var.b_ ## bit
```

will transform:

```
BIT(port, 3) = 1;
```

into

```
port.b_3 = 1;
```

Without this operator, it would have been impossible to get rid of the white space between the base name and the bit number, and the compiler would have been unable to get the proper syntax.

Once a symbol has been completely replaced, the resulting string is scanned again to look for subsequent replacements. Note that the original symbol will not be expanded again when rescanning the first result thus avoiding recursive endless behaviour.

An ANSI macro cannot create a new preprocessor directive. The COSMIC compiler allows such a feature by starting the replacement list with a **\#** string. Once expanded, such a line will be re-executed as a preprocessor directive.

```
#define INC(file) \#include #file ".h"

INC(stdio)
```

will expand to

```
\#include "stdio.h"
```

and will be re-executed after the removal of the prefixing **** thus including the file `stdio.h`.

A C99 extension allows the last parameter of a macro definition to be the **...** punctuator, allowing the macro to accept a variable number of arguments. In the replacement text, the predefined **__VA_ARGS__** symbol will be replaced by the complete list of extra arguments matching the **...** parameter.

```
#define display(fmt, ...)   printf(fmt, __VA_ARGS__)

     display(“Error %d\n”, errnum);
     display(“File %s line %d\n”, filename, linenum);
```

will expand to:

```
     printf(“Error %d\n”, errnum);
     printf(“File %s line %d\n”, filename, linenum);
```

Hazardous Behaviours

It is important to keep in mind that this replacement is done only on a text basis, without any attempt to understand the result. This may lead in a few unexpected side effects.

The following macro is used to get the absolute value of an expression:

```
#define ABS(x)   x > 0 ? x : -x
```

and as it is written, it is correct. In the following usage:

```
a = ABS(-b);
```

the replacement will produce:

```
a = -b > 0 ? -b : --b;
```

obtained directly by replacing **a** by **-b**. The last expression creates a **--b** expression which decrements the **b** variable instead of loading its direct value. To avoid such a situation, it is recommended to enclose any occurrence of any parameter by parentheses in the replacement list:

```
#define ABS(x)  (x) > 0 ? (x) : -(x)
```

The replacement then becomes:

```
a = (-b) > 0 ? (-b) : -(-b);
```

and now, the last expression will be evaluated as **b** negated twice, and optimized in a direct load of **b**.

Another side effect may be produced by an unexpected concatenation. The macro:

```
#define SUM(a, b)   (a) + (b)
```

may be used in the following expression:

```
x = SUM(y, z) * 2;
```

When expanded, it becomes:

```
x = (y) + (z) * 2;
```

and now, the priority rules change the expected behaviour to:

```
x = (y) + ((z) * 2);
```

The solution is simply to use parentheses around the whole definition:

```
#define SUM(a, b)  ((a) + (b))
```

will produce as a result:

```
x = ((x) + (y)) * 2;
```

and the macro expansion is protected against any other operator.

A last example of an unexpected behaviour uses increment operators. Assuming the previous definition for the **`ABS`** macro, the usage:

```
x = ABS(*p++);
```

will expand to:

```
x = (*p++) > 0 ? (*p++) : -(*p++);
```

In this expression, the pointer will be incremented twice and the result is wrong coming from the element following the one tested. Unfortunately, there is no syntax trick to avoid this one.

Most of these errors are very difficult to find, because they do not produce errors at compile time, and because you do not see what is actually expanded in reading the C source. By reading the example getting the absolute value of **`*p++`**, there is only one increment seen. The extra one is implied by the macro expansion, but you have to look at the macro definition to find that. That is why it is important to immediately check that a name is a macro, this is made easier using only uppercase names for macros as a convention.

It is possible to have a look at the expanded source file by compiling it with the **-sp** option, producing a result in a file with a **.p** extension.

Predefined Symbols

The preprocessor predefines a few symbols with a built-in behaviour. Those symbols cannot be undefined by a *#undef* directive and then cannot be redefined to any other behaviour.

`__FILE__`	expands to a text string containing the name of the file being compiled.
`__LINE__`	expands to a numerical value equal to the current line number in the current source file.
`__DATE__`	expands to a text string containing the date you compiled the program. The date format is "`mmm dd yyyy`", where `mmm` is the month abbreviated name, `dd` is the day and `yyyy` the year.
`__TIME__`	expands to a text string containing the time you compiled the program. The time format is "`hh:mm:ss`", where `hh` is the hours, `mm` the minutes and `ss` the seconds.
`__STDC__`	expands to the numerical value 1 indicating that the compiler implements the ANSI features.

The COSMIC compiler also defines the following symbols:

`__CSMC__`	expands to a numerical value whose each bit indicates if a specific option has been activated:

bit 0: set if nowiden option specified (**+nowiden**)
bit 1: set if single precision option specified (**+sprec**)
bit 2: set if unsigned char option specified (**-pu**)
bit 3: set if alignment option specified (**+even**)
bit 4: set if reverse bitfield option specified (**+rev**)
bit 5: set if no enum optimization specified (**-pne**)
bit 6: set if no bitfield packing specified (**-pnb**)
bit 7: set if no constant replacement specified (**-pnc**)
bit 8: set if no constant propagation specified (**-pcp**)
bit 9: set if prototype checking specified (**+proto**)
bit 10: set if hardware FPU selected (**+fpu**)

This extra symbol may be used to select the proper behaviour depending on the compiler options. Some bits may not be significant depending on the target processor.

`__LGLG__` expands to a numerical value equal to 1 if **long long** type is supported, and 0 otherwise.

`__TRGT__` expands to a text string containing the target processor name.

`__VERS__` expands to a text string containing the compiler version.

Conditional Directives

The conditional directives are:

```
#ifdef IDENT

#ifndef IDENT

#if expression
```

and are associated with the ending directives

```
#else

#endif

#elif expression
```

A conditional directive is always followed by an ending directive. All the C lines enclosed by the conditional directive and its ending directive will be compiled or skipped depending on the result of the condition test.

`#ifdef IDENT`	is true if there is the macro **`IDENT`** has been previously defined
`#ifndef IDENT`	is true if **`IDENT`** is not the name of a macro previously defined
`#if expression`	is true if the result of expression is not zero

The expression will be evaluated as a constant expression, so after all the possible macro replacements inside the expression, any word which has not been replaced by a number or an operator is replaced by the value zero before the evaluation. The COSMIC compiler accepts the special operator `sizeof` and `enum` members inside a `#if` expression, although this is not supported by the ANSI standard.

An ending directive may also become a conditional directive starting a new conditional block, such as `#else` and `#elif`.

Here are a few possible constructs:

```
#ifdef DEBUG
printf("trace 1\n");
#endif
```

If the symbol **`DEBUG`** has been defined previously, the line *printf...* will be compiled. Otherwise, it is simply skipped.

```
#if TERM == 1
init_screen();
#else
init_printer();
#endif
```

If the symbol **`TERM`** has been previously defined equal to **`1`**, the line **`init_screen();`** is compiled, and the line **`init_printer`** is skipped. If **`TERM`** is define to anything else, or if **`TERM`** is not defined, the behaviour is the opposite (if **`TERM`** is not defined, it is replaced by **`0`** and the expression **`0 == 1`** is false).

The **`#elif`** directive is simply a contraction of a **`#else`** immediately followed by a **`#if`**. It avoids too complex embedding in case of multiple values:

```
#if TERM == 1
...
#elif TERM == 2
...
#elif TERM == 3
...
#else
...
#endif
```

Control Directives

The control directives are:

#include

```
#include "filename"
```

or

```
#include <filename>
```

The preprocessor replaces such a line by the full content of the file whose name is specified between double quotes **“** or angle brackets < and >. Files are searched first in the current directory, then in some predefined system directories, or user specified directories. An error will occur if the file is not found in any of the specified directories. An included file may contain other `#include` directives.

#error

```
#error rest_of_the_line
```

If this directive is encountered, the compiler outputs an error message whose content is the `rest_of_the_line`. This directive is interesting to force an error if something is detected wrong in the defined symbols:

```
#ifndef TERM
#error missing definition for TERM
#endif
```

If the symbol `TERM` is not defined, the compiler will output an error message containing the text “`missing definition for TERM`”, and will fail to compile the source file.

#line

```
#line number "filename"
```

This directive redefines the current line number to the specified number, and the file name to the specified name in the text string. This is mainly used by automatic code generators to allow an error to refer to the input file name and line number rather than the intermediate C source file produced. This is almost never used by a human written program. Note that this directive modifies the value of the predefined symbol `__FILE__`.

#pragma

```
#pragma rest_of_the_line
```

This directive allows passing to the compiler any configuration directive useful for code generation. There is no standard or predefined syntax for the content of the directive, and each compiler may implement whatever it needs. The only defined behaviour is that if the compiler does not recognize the directive, it skips it without error message, thus keeping this directive portable across different compilers.

The COSMIC compiler implements two pragmas to control the allocation of objects in memory spaces and in assembler sections.

Space Definition

```
#pragma space <class> <kind> <modifiers>
```

The **#pragma space** allows you to choose in which memory space C objects are allocated. Such a directive will affect the objects declared after it, until a new directive changes again the configuration.

The <*class*> field contains a keyword describing the object class:

`extern`	for global objects
`static`	for static objects
`auto`	for local objects
`*`	for pointed objects
`const`	for compiler constants

If this field is empty, all the classes are selected, except **const**.

The <*kind*> field specifies which kind of object is selected:

`[]`	specifies variables
`()`	specifies functions

If this field is empty, both kinds are selected.

The <*modifier*> field contains a list of modifiers to be applied by default to the objects selected. Each modifier starts with the `@` character and must be a valid modifier supported by the compiler, as each target supports a different set of modifiers. If the *<modifier>* field is empty, all the attributes are turned off.

The following directive:

```
#pragma space extern () @far
```

turns all the following global functions to be `@far` (bank switched).

Static functions are not affected by this directive.

```
#pragma space [] @eeprom
```

turns all the following variables of any class to be flagged as being in eeprom. The effect of such a directive will be cancelled by the following:

```
#pragma space []
```

Refer to the specific compiler manual for the list of supported space modifiers.

Section Redefinition

```
#pragma section <modifier> <kind_and_name>
```

The **#pragma section** directive allows the compiler to modify the section in which objects are allocated. The compiler splits the various program components in the following default sections:

executable code	`.text`
constants	`.const`
initialized variables	`.data` or `.bsct`
uninitialized variables	`.bss` or `.ubsct`
eeprom variables	`.eeprom`

Variables are allocated in the **.bsct** or **.ubsct** when flagged as **zero page**. Those sections and the **.eeprom** section may not be defined depending on the target capabilities.

Each of these sections can be renamed. The compiler then creates a new assembler section with the proper name and attributes, and produces there the matching objects. The compiler will prepend a dot `.` to the provided name, and will check that the final name is not longer than **14** characters.

The name is provided in the <*kind_and_name*> field along with the kind of object in such a way:

`( name )`	defines a name for executable code
`{ name }`	defines a name for initialized variables
`[ name ]`	defines a name for uninitialized variables

The <*modifier*> allows choosing the right section by specifying the proper attribute, depending on the object kind selected:

`const { name }`	changes the **.const** section name
`@tiny { name }`	changes the **.bsct** section name (or **@dir** depending on the actual target)
`@eeprom { name }`	changes the **.eeprom** section name

Note that `[]` may be used instead of `{}` with the same effect for sections which are not sensitive to the initialization or not (**.const** and **.eeprom**).

If the name part is omitted between the parentheses, the matching section name turns back to its default value.

In the following example:

```
#pragma section {mor}
char MOR = 0x3c;
#pragma section {}
```

The compiler creates a section named `.mor` which replaces the default **.data** section. The variable `MOR` is then created in the `.mor` section. The compiler reverts to the original *.data* section for the next initialized variables.

The interrupt vectors can be located in a separate section with the following code:

```
#pragma section const {vector}
void (* const vectab[])(void) = {it1, it2, it3};
#pragma section const {}
```

The `vectab` table will be produced in the created **.vector** section instead of the default *.const* section.

Loop Unrolling

The COSMIC compiler allows controlling loop unrolling wherever this feature is available. A loop can be unrolled if the compiler can detect without any ambiguity the control variable and the iteration count. The compiler can then replace a loop by the repetition of the loop body as many times as the loop is iterated, replacing the control variable by its equivalent constant index value in each repetition. A loop is unrolled if the total amount of C operators in the unrolled code is not larger than a limit provided by the directive:

```
#pragma unroll <num>
```

where `<num>` is a number defining that maximum limit. This limit is expressed in number of C operators rather than code size because the compiler has no idea yet of how much code will be produced in that loop. The default value is 0, so loop unrolling is disable by default.

Space modification

The COSMIC compiler allows redefining the space attribute by a directive, which can be placed outside the code itself. This feature is very convenient when porting code which cannot be modified or when tuning the code efficiency without changing it. This feature is controlled by the directive:

```
#pragma attribute <declaration>
```

where `<declaration>` is a restricted C object declaration specifying the necessary space attributes. Basic types can be omitted, but compound specifiers must be specified (`[]` for arrays, `*` for pointers, `()` for functions). For functions, arguments must be specified if they also contain space modifiers.

The effect of such directive is to add the specified space modifiers when parsing the same object in the code itself. The compiler will detect any conflict with any already specified modifiers and will issue an error message in such a case.

The following directives:

```
#pragma attribute @far var;
#pragma attribute @near *ptr;
#pragma attribute @tiny arr[];
```

applied to such variable declarations:

```
int var;
char *ptr;
long arr[10];
```

will behave as if the objects were explicitly declared:

```
@far int var;
@near char *ptr;
@tiny long arr[10];
```

Such attribute directives can be placed into a separate header file, which can be included automatically in each compiled file by using the **-ph** compiler option.

Inline Assembler

The **#pragma asm** and **#pragma endasm** directives allow assembly code to be directly inserted in a C program. Assembly code is entered between those two directives as if they were written in an assembler program. Note that there is no direct connection possible with existing C objects. When used outside a function, such a block behaves syntactically as a declaration. When used inside a function, such a block behaves as a *block statement*. It is not necessary to enclose it with curly braces {} although it is more readable. The compiler accepts the directives **`#asm`** and **`#endasm`** as shortcuts for the **`#pragma`** ones.

By default, the assembly lines entered between the **#asm** and **#endasm** directives are not preprocessed. It is possible to apply the preprocessor #defines to the assembly text by specifying the **-pad** compiler option.

APPENDIX
A

Implementation Defined Behavior

This appendix describes how COSMIC handles the implementation-defined areas of the C standard.

COSMIC compilers are *Cross* compilers and conform to the *Freestanding* environment of the ANSI/ISO standard.

Description of Implementation-Defined behavior

This appendix follows the same order as the ISO appendix. Each covered item includes references to the ISO standard chapter and section in **(parenthesis)** and to the description in this manual when available (see *paragraph*).

Translation

Diagnostics are produced in the form:

#error <pass> <file>: <line_number>(char_pos) <error message>

where <*pass*> is the name of compiler component, <*file*> is the name of the source file in which the error was encountered, <*line_number*>, the line number in the source file, (*char_pos*) the character position in the line optionally followed by a + sign and the word length, and <*error message*> is the explanation message. See the error messages in the "*C Cross Compiler User's Guide, Appendix A*". **(5.1.1.3)**.

Warnings when enabled are displayed with the same format starting with *#warning*.

Environment

- **Arguments to main**. The function called at program startup is called **main**. **(5.1.2.2.2.1)**. There is no prototype declared for **main** and its only definition is:

  ```
  void main(void)
  ```

- **Interactive devices**. There are no interactive devices **(5.1.2.3)**.

Identifiers

- All identifiers are significant up to 255 characters and are case-sensitive (see "*Identifiers*" in **Chapter 2**). **(6.1.2)**.

Characters

- The standard ASCII character set is used for both source and execution environments. **(5.2.1)** with **8** bits per character **(5.2.4.2.1)**.
- Source characters are mapped identically in the execution environment. **(6.1.3.4)**.

- Character constants containing more than one character are evaluated as a number expressed in base 256 (see "*Constants*" in **Chapter 4**). **(6.1.3.4)**.
- Multibyte characters and strings are not implemented. **6.1.3.4)**.
- A plain char type is defaulted to *unsigned char*. This behavior is tunable (see "*Integer Types*" in **Chapter 2**) **(6.2.1.1)**.

Integers

- Integers are using the two's complement representation (see "*Integer Types*" in **Chapter 2**). **(6.1.2.5)**.
- Demoted integers are truncated. **(6.2.1.2)**.
- Bitwise operations on signed integers work the same as unsigned, but the result is signed (see "*Operators*" in **Chapter 4**). **(6.3)**.
- The remainder of a signed division has the same sign as the dividend (see "*Operators*" in **Chapter 4** **(6.3.5)**.
- The right shift of a negative integer from an integral type preserves the sign-bit (see "*Bitwise operators*" in **Chapter 4**). **(6.3.7)**.

Floating point

- Floating-point numbers are represented using the IEEE-754 floating point format (see "*Real Types*" in **Chapter 2**). **(6.1.2.5)**.
- When an integer is converted to a floating-point number, which cannot be represented exactly, the results is rounded down. **(6.2.1.3)**.
- When a floating-point number is converted to a narrower floating-point number, which cannot be represented exactly, the result is rounded down. **(6.2.1.4)**.

Arrays and Pointers

- The *size_t* type is equivalent to int. **(6.3.3.4, 7.1.1)**.
- An **@tiny** pointer is equivalent to an *unsigned char*, an **@near** pointer is equivalent to an *unsigned int* and an **@far** pointer is equivalent to an *unsigned long*. **(6.3.4)**.
- Subtracting **@tiny** or **@near** pointers produces an *int* result, subtracting **@far** pointers produces a *long* result (see "*Arithmetic operators*" in **Chapter 4**). **(6.3.6, 7.1.1)**.

Registers

Register variables are available depending on the target processor. See "*Register Usage*" in the "*C Cross Compiler User's Guide*". **(6.5.1)**.

Structures, Unions, Enumerations, and Bit-fields

- If a member of union is accessed using a member of a different type, the result will be memory dependant. **(6.3.2.3)**.
- Padding and alignment of members of structures are depending on the target processor and follows the same alignment constraints than simple variables (see "*Structures*" in **Chapter 2**). **(6.5.2.1)**.
- A plain **int** bit-field is treated as an **unsigned int** bit-field (see "*Structures*" in **Chapter 3**). **(6.5.2.1)**.
- Bitfields are filled from the less significant bit to the most significant bit by default. This behavior is tunable (see "*Structures*" in **Chapter 3**). **(6.5.2.1)**.
- The bitfield allocation is described in "*Structures*" in **Chapter 3** **(6.5.2.1)**.
- An enumeration type is implemented as a **char**, a **short** or a **long** depending on the members value range (see "*Enumerations*" in **Chapter 3**). **(6.5.2.2)**.

Qualifiers

Access to a **volatile** object is never optimizing out by the compiler (see "*Modifiers*" in **Chapter 3**). **(6.5.5.3)**.

Declarators

There is no limitation to the number of declarators in types (see " *Declarations*" in **Chapter 3**) **(6.5.4)**.

Statements

There is no limitation to the number of *case* values in a *switch* statement (see " *Switch statement*" in **Chapter 5**). **(6.6.4.2)**.

Preprocessing Directives

- Character constant in preprocessor expressions are getting the same value than in the execution environment and can then be negative (see "*Conditional Directives*" in **Chapter 6**). **(6.8.1)**.

- Access method to included source files is described in “*#include*” in **Chapter 6**. **(6.8.2)**.
- **#pragma** directives are described in “*#pragma*” in **Chapter 6**. **(6.8.6)**.
- Definition for **__DATE__** and **__TIME__** is described in “*Predefined Symbols*” in **Chapter 6**. **(6.8.8)**.

APPENDIX
B

Supported C99 Extensions

This appendix describes which C99 features are implemented in the COSMIC compiler.

Description of Supported C99 Extensions

This appendix describes the features supported by the compiler as C99 extensions when compiling with **-pc99** compiler option. Some C99 extensions are supported unconditionally like the // comment, the **_Bool** and the **long long** types whenever implemented.

Trailing comma in enumeration declaration

The last member of an enumeration list may be followed by a trailing comma before the closing brace. The comma will be ignored.

Flexible array

The last field of a structure may be an array declared without any explicit dimension.

Repeated qualifiers

The **const** and **volatile** qualifiers may be repeated in the same declaration either directly or indirectly through a typedef declaration.

Inline keyword

The compiler recognizes the **inline** keyword as a function storage class requesting the function to be inlined. It behaves like the **@inline** modifier.

Variable macro arguments

The last parameter of a macro definition may be the **...** punctuator allowing a macro to have a variable number of arguments. The predefined **__VA_ARGS__** symbol is replaced by the full list of extra arguments matching the **...** parameter.

Index

Symbols

#asm 6-91
#elif directive 6-84
#else directive 6-84
#endasm 6-91
#endif directive 6-84
#error directive 6-86
#if directive 6-84
#ifdef directive 6-84
#ifndef directive 6-84
#line directive 6-86
#pragma
  asm 6-91
  attribute 6-90
  directive 6-87
  endasm 6-91
  section 6-88
  space 6-87
#undef directive 6-76
@far modifier 4-46
@far space modifier 3-21
@inline modifier 3-24
@near modifier 4-46
@near space modifier 3-21
@nostack modifier 3-23
@packed modifier 3-23
@tiny modifier 4-46
@tiny space modifier 3-21
\ character 2-5
_asm function 4-51
_Bool type 3-19
_Fract 2-12, 3-18
_Fract type 2-12, 4-39

A

address 2-12, 3-33
address constant 3-19
address return 4-49
and operator 5-62
and, bitwise 4-44
argument list 3-28
array 2-13
array initialization 3-20
array keyword 3-20
array of arrays 3-21
arrays of pointers 3-21
Assignment operators 4-46
auto keyword 3-31

B

bank switch 2-13
bit variable 3-19, 3-33, 4-47
bitfields 2-14
bitwise operators 4-44
block 3-30
block statement 5-60
body of a function 3-30
body statement 5-67
Boolean operators 4-45
break statement 5-67, 5-70

C

C expressions 4-35

C identifier 4-36
C keywords 2-7
C operators 2-8
C program 2-9
C punctuators 2-8
C99 extension 1-4
  _Bool type 2-11
  inline keyword 3-24
  long long type 2-9
  supported features A-100
  variable macro arguments 6-79
case label 5-69
cast operator 4-52
comment
  end 2-6
  start 2-6
Comments 2-6
conditional
  directives 6-75
  features 6-75
conditional operator 4-51
configuration directive 6-87
const
  keyword 3-21
  modifier 3-22
constant 2-8
  object 3-22
constants 2-7, 4-36
continue statement 5-68
control directives 6-75, 6-86
current line number 6-86

D

declaration 2-9
declaration, of an object 3-17
decrementation 4-48
default label 5-70
do statement 5-64
double 3-18
double type 4-39

E

eeprom 6-88
else keyword 5-62
else statement 5-62
enum keyword 3-27
enumeration 2-10, 2-14, 3-27
error message 6-86
escape sequence 4-37
exclusive or, bitwise 4-44
expression 4-35, 5-60
extern keyword 3-30

F

false, logical value 4-45
field 2-14
file scope 3-31
first expression 5-65
flexible array 3-26
float 2-12
float type 3-18, 4-39
for statement 5-65, 5-66
fractional
  suffix 4-39
  type 2-12
  variable 3-18
function 2-14, 3-28
function body 5-59
function call operator 4-49

G

global variable 3-32
goto label 5-71
goto statement 5-71

I

identifier 2-7
identifiers 2-7
if statement 5-61, 5-62
index 2-13
indexing operator 4-49
initial value 3-17

initialization
    of a function 3-30
    of an array 3-20
inline keyword 3-24
int 2-9, 3-18
int type 4-38
integer type 2-9
integer variable 3-18

K
K&R syntax 3-29
Kernigan and Ritchie syntax 3-28
keyword 2-7
keyword name 2-7
keywords 2-7

L
label 5-71
leave, a function 5-72
line terminator 2-5
load instruction 5-61
local variable 3-33
local variables 3-30
logical true 4-45
long 2-9
long _Fract 2-12, 3-18
long _Fract type 4-39
long double 2-11, 3-18
long double type 4-39
long enumerations 3-28
long int 2-9
long long 2-9, 3-18
long long type 4-38, 6-83
loop
    body 5-65
    initialization 5-65
    iteration 5-65
    unroll 6-90
lowercase 2-7
L-value expression 4-46

M
macro 6-75
macro directives 6-75
main 2-15
memory model 2-13
memory modification 4-46
memory space 6-87
modifier 3-21

N
name 3-17
nop instruction 5-60
not operator 5-67
numerical constant 3-19

O
object class 6-87
operands 4-42
operator 2-8, 4-49
Operators 4-42
operators 2-7
or operator 5-62
or, bitwise 4-44

P
parameters 3-29
peripheral registers 5-61
physical register 3-31
pointer 2-12
pointer keyword 3-19
pointer return 4-48
pointer to void 4-46
pointers to arrays 3-21
pointers to pointers 3-21
post incrementation 4-48
pre incrementation 4-48
predefined space modifiers 3-21
predefined symbol
    __CSMC__ 6-82
    __DATE__ 6-82
    __FILE__ 6-82

  __LGLG__ 6-83
  __LINE__ 6-82
  __STDC__ 6-82
  __TIME__ 6-82
  __TRGT__ 6-83
  __VA_ARGS__ 6-79, A-100
  __VERS__ 6-83
preprocessing directives 2-6
preprocessor 6-75
preprocessor directive 6-75
prototyped syntax 3-28
punctuation 2-7
punctuators 2-8

R

real variable 3-18
register class 5-60
register keyword 3-31
register object 3-32
register variable 3-31
registers modification 4-46
return 5-72
return statement 5-72
returned value type 3-28
right shift operator 4-44
R-value expression 4-46

S

scope, of variable 3-30
second expression 5-65
semicolon 5-60
sequence operator 4-52
set of variables 4-35
short 2-9
short _Fract 2-12, 3-18
short int 2-9
signed 2-10
signed keyword 3-18
size of a pointer 2-13
sizeof type 4-41
space
  attribute 6-90
  modifiers 3-21
special ponctuator 3-29
Statements 5-59
static keyword 3-31, 3-32
stop compiler optimizing 3-22
storage class 3-17, 3-30
string constant 4-40
struct keyword 3-24
structure 2-13, 3-24
  array field 3-26
  initialization 3-25
switch statement 5-69

T

tag name 3-24
terminator character 5-59
then keyword 5-62
third expression 5-65
true, logical value 4-45
type 3-17
type equivalent 3-32
typedef keyword 3-32

U

union 2-14, 3-27
unsigned 2-10
unsigned char 3-18
unsigned int 3-18, 4-38
unsigned keyword 3-18
unsigned short 2-13
uppercase 2-7
useless expression 5-61

V

variable number of arguments 3-29
Variables 4-36
void 5-72
volatile
  attribute 5-61
  keyword 3-21
  modifier 3-22

W

while statement 5-63
wide
    character 4-38, 4-41
    string 4-41