LAPIS SEMICONDUCTOR

SQ003023E092

CCU8
Language Reference

Program Development Support Software

ISSUE DATE: Oct. 2011

Table Of Contents

1. PREPROCESSOR

1.1 INTRODUCTION

CCU8 may be invoked with /LP or /PC option so as to process text without compiling. CCU8 behaves like a text processor that manipulates the text of a source file, when invoked with /LP or /PC option.

Preprocessor performs the following functions

1. Macro substitution
2. Conditional compilation
3. File inclusion
4. Line control
5. Error generation
6. Mixed Language programming
7. Other implementation dependent actions(Using pragmas).
8. Trigraph Sequences replacement.

Lines beginning with a #, perhaps preceded by white space, communicate with the preprocessor. The syntax of these lines is independent of the rest of the language. Line boundaries are significant. End of file must not occur in a preprocessor directive line. Preprocessor directives may appear anywhere in a file. However, they apply only to the remaining part of the source file in which they appear.

1.2 TRANSLATION PHASES

Preprocessing will be performed by the following four translation phases in the given order:

1. Trigraph sequences are replaced by the corresponding single-character internal representations.
2. Each instance of a new-line character and an immediately preceding backslash character is deleted, splicing physical source lines to form logical source lines.
3. The source file is decomposed into preprocessing tokens and sequences of white-space characters (including comments).
4. Preprocessing directives are executed and macro invocations are expanded. A #include preprocessing directive causes the named headers or source file to be processed from phase 1 to phase 4, recursively.

1.2.1 Trigraph sequences

All occurrences in a source file of the following sequences of three characters (called trigraph sequences) are replaced with the corresponding single character.

Trigraph Sequence	Replacement character
??=	#
??(	[
??/	\
??)	]
??’	^
??<	{
??!	|
??>	}
??-	~

Each ‘?’ that does not begin one of the trigraphs listed above is not changed.

Example 1.1

INPUT:

```
main ()
??<
??>
```

OUTPUT:

```
main ()
{
}
```

1.2.2 Line Splicing

A new line character and an immediately preceding backslash character are deleted, and line following the new-line character is considered as continuation of the previous line.

1.3 MACROS

1.3.1 Introduction

Macro is a facility that enables user to assign a symbolic name to a sequence of tokens. The symbolic name can then be used in the source file to represent the sequence of tokens.

The following two preprocessor directives facilitate in macro definition and deletion.

a) #define
b) #undef

Macro expansion is a text processing function that replaces the macro name with the corresponding token sequences.

Parameters may also be defined to represent arguments passed to the macro. The replacement text of a macro with arguments may vary for different calls. The following two special operators influence the replacement process.

a) stringizing (#)

b) token pasting (##).

1.3.2 Macro Definition

1.3.2.1 Defining Macros Without Parameters

Syntax :

*# **define** identifier token_sequence*

The #define directive causes the preprocessor to replace subsequent occurrences of the identifier with the given sequence of tokens.

Leading and trailing white spaces around the token sequence are discarded.

The macro name must be a valid ‘C’ identifier. The token sequence represents the replacement text.

Example 1.2

```
# define ABC          1 + 2
```

MACRO CALL	REPLACEMENT TEXT
ABC + 3	1 + 2 + 3
fn(ABC)	fn(1 + 2)

1.3.2.2 Defining Macros With Parameters

Syntax :

*# **define** identifier([parameter_list]) token_sequence*

A macro definition is assumed to have parameters when there is no space between the identifier and the open parenthesis ‘(’.

The parameter list is optional. If present, it consists of one or more parameters. The parameter list must be enclosed within parentheses. Each parameter must be an unique identifier in the parameter list. Adjacent parameters are separated by a comma.

Parameters appear in the token sequence to mark the places where arguments must be substituted. However, the same parameter may occur more than once in the token sequence.

Leading and trailing white spaces around the token sequence are discarded.

Example 1.3

```
# define ABC(x,y)        x + y
```

MACRO CALL	REPLACEMENT TEXT
ABC (1,2)	1 + 2
ABC (2,x)	2 + x

1.3.2.3 Operators In Macro Processing

1.3.2.3.1 Stringizer

Operator symbol : #

Syntax :

# parameter

The stringizer is used only in macros defined with parameters. It may occur in the token sequence. A parameter must follow the stringizer.

During expansion, the argument is enclosed within quotation marks and treated as a string literal.

A \ character is inserted before each **“** and \ character of a character constant or string literal (including the delimiting **“** characters).

Example 1.4

```
# define A(b)        #b
# define X(y)        (#y “\n”)
```

MACRO CALL	REPLACEMENT TEXT
A(1)	“1”
X(abc)	(“abc” “\n”)
A(“a\c”)	“\”a\\c\””
A(“abcde\n”)	“\”abcde\\n\””

1.3.2.3.2 Token Paster

Operator symbol : ##

Syntax :

token ## token

Token paster is used in macros defined with or without parameters.

Token paster operator concatenates adjacent tokens, deleting white space between them, to form a new token.

Token paster cannot occur at the beginning and end of the token sequence.

Example 1.5

```
# define A(b,c)        b ## c
# define X(a)          a ## 1
# define Y(a)          1 ## a
# define ONE           12 ##4
```

MACRO CALL	REPLACEMENT TEXT
A(1,2)	12
X(34)	341
Y(43)	143
ONE	124

1.4 MACRO EXPANSION

1.4.1 Expansion Of Macros Without Parameters

The subsequent instances of the identifier, defined as a macro without parameters, causes the preprocessor to replace the instances of the identifier with the given sequence of tokens.

Example 1.6

```
# define ONE        1
# define TWO        2
```

MACRO CALL	REPLACEMENT TEXT
x = ONE + TWO + ONE	x = 1 + 2 + 1

The replaced token sequence is repeatedly rescanned for more defined identifiers.

Example 1.7

# define ONE	THREE
# define TWO	2
# define THREE	3
MACRO CALL	REPLACEMENT TEXT
x = ONE + TWO + THREE	x = 3 + 2 + 3

A replaced identifier is not replaced if it turns up again during rescanning. Instead it is left unchanged to avoid recursion.

Example 1.8

# define ONE	TWO
# define TWO	THREE
# define THREE	TWO
MACRO CALL	REPLACEMENT TEXT
x = ONE + TWO + THREE	x = TWO + TWO + THREE

During expansion, each collection of white spaces is replaced by a single blank.

Example 1.9

# define ABCD	a + b + c+d
# define XYZ	a/* abcde */+ 2
MACRO CALL	REPLACEMENT TEXT
ABCD	a + b + c+d
XYZ	a + 2

The macro identifiers within quotation marks are not considered as a macro call.

Example 1.10

# define ONE	1
MACRO CALL	REPLACEMENT TEXT
“ONE”	“ONE”

1.4.2 Expansion Of Macros With Parameters

Identifiers defined as a macro with parameters may be called by writing

identifier[white space]([actual_argument_list])

Example 1.11

```
# define ADD(a,b)                  a + b
# define MUL(a,b)                  (a * b)

MACRO CALL                         REPLACEMENT TEXT

x=MUL(23,43) - ADD(12,400)         x=(23 * 43) - 12 + 400
```

1.4.3 ARGUMENTS OF MACRO CALL

The arguments of a call are comma separated token sequence. Commas that are enclosed within quotes or parentheses do not separate arguments.

The number of arguments in the call must match the number of parameters in the definition.

Leading and trailing spaces in each argument are discarded.

Collection of white spaces within an argument is replaced by a blank.

The arguments may run through more than one line.

Example 1.12

```
# define ADD(a,b)       a + b
# define MUL(a,b)       (a * b)

MACRO CALL              REPLACEMENT TEXT

ADD(1,2)                1 + 2
MUL(12,2)               (12 * 2)
ADD(xxx(1,2),3)         xxx(1,2) + 3
ADD(1    + 2,3)         1 + 2 + 3
ADD (    11,3 )         11 + 3
ADD (12345 +
678, 9)                 12345 + 678 + 9
```

The tokens in the arguments are examined for macro calls, and expanded as necessary, before expanding the call. However, if the argument is preceded by #, or preceded or followed by ##, the outer call is expanded first.

Example 1.13

```
# define ADD(a,b)            a + b
# define MUL(a,b)            (a * b)
```

MACRO CALL	REPLACEMENT TEXT
ADD(MUL(1,2),3)	(1 * 2) + 3
MUL(MUL(ADD(1,2),3),4)	((1 + 2 * 3) * 4)

If the parameter in the replacement sequence is preceded by #, the argument tokens are not examined for macro calls.

Example 1.14

```
# define ONE(a)        #a
# define TWO(a,b)      (a * b)
```

MACRO CALL	REPLACEMENT TEXT
ONE (TWO (1,2))	“TWO (1,2)”

If the parameter in the replacement sequence is preceded or followed by a ##, the tokens in the argument tokens are not examined for macro calls.

Example 1.15

```
# define CAT(a,b)      a ## b
```

MACRO CALL	REPLACEMENT TEXT
CAT (CAT(1,2),3)	CAT(1,2)3

In the above example, the presence of ## prevents the arguments of the outer call from being expanded first. Hence the expansion of outer call results in CAT(1,2)3. The identifier CAT in the replacement text is not expanded, since it turns up again.

Example 1.16

```
# define CAT(a,b)      a ## b
# define TWO(a,b)      (a + b)
```

MACRO CALL	REPLACEMENT TEXT
CAT (TWO(1,2),3)	(1 + 2)3

In the above example, the identifier CAT is expanded first, before the expansion of the arguments and the result is (1+2)3. The identifier TWO in the replacement text is expanded as it has not been expanded already.

Example 1.17

```
# define CAT(a,b)        a ## b
# define XCAT(a,b)       CAT(a,b)
```

<u>MACRO CALL</u>	<u>REPLACEMENT TEXT</u>
XCAT(XCAT(1,2),3)	123

In the above example, the token sequence of XCAT does not contain ##, the arguments is expanded first. Therefore, the inner call XCAT(1,2) is expanded as 12 and then the outer call XCAT (12,3) is expanded as 123.

1.5 MACRO REMOVAL

Syntax :

*# **undef** identifier*

The #undef directive causes the preprocessor to remove the definition of the identifier. Subsequent occurrences of the identifier are ignored by the preprocessor till it is defined again.

To remove an identifier specified as a macro with parameters, only the identifier has to be specified in the #undef directive.

If the identifier specified has not been previously defined using the #define directive, no error message is displayed. This ensures that the identifier is undefined.

Example 1.18

```
# define ONE          1

x = ONE ;

# undef               ONE

y = ONE

# define A(b,c)       b + c
# undef               A
```

In the above example, the variable x is assigned a constant 1, whereas the variable y is not assigned the constant value 1.

1.6 REDEFINITION OF MACROS

Redefinition of the macro is erroneous, unless the redefinition satisfies the following.

a) The token sequence must be identical
b) If the identifier is defined as a macro with parameters, the number of parameters must be equal.

The following redefinitions are erroneous.

Example 1.19

```
# define A              1+2
# define A              1-2
```

Example 1.20

```
# define A              1+2
# define A              1 + 2
```

Example 1.21

```
# define A              12
# define A(b)           12
```

Example 1.22

```
# define A(b)           b
# define A              b
```

Example 1.23

```
# define A(one,two)     one + two
# define A(one)         one + two
```

The following redefinitions are nonerroneous.

Example 1.24

```
# define A              1 + 2
# define A              1 + 2
```

Example 1.25

```
# define A(one,two)     one + two
# define A(one,two)     one + two
```

Example 1.26

```
# define A(one, two)    one + two
# define A(x,y)         x + y
```

1.7 FILE INCLUSION

1.7.1 Introduction

The **#include** directive causes the preprocessor to replace the line where the directive has occurred by the entire contents of the specified file during compilation.

File inclusion makes it easy to handle collection of **#define** statements and declarations(among other things). They are often kept in a separate file and read into the ‘C’ source file at compile time. In this way, libraries of different macros may be used in many different source files.

If the file specified in the directive is not present, fatal error is displayed and the compilation is terminated.

Nesting level of include files is restricted to ten files (including the source file).

1.7.2 Include File Specification Using Double Quotation Marks

Syntax :

*# **include** “filename”*

The filename may contain a path specification. If the filename is not specified with the path, it is searched in the following order :

a) directory of parent files is searched (parent file is the file containing the # include directive)
b) the directories of any grand parent files
c) the directories specified using /I command line option
d) the standard directory set by the environment variable INCLU8.

Example 1.27

```
# include “\dir1\dir2\abc.c”
```

The above **#include** directive causes the compiler to replace the contents of the file abc.c present in the directory \dir1\dir2 for the line where the directive is specified.

1.7.3 Include File Specification Using Angle Brackets

Syntax :

***# include** <filename>*

The filename may contain a path specification.

If the filename is not specified with the path, it is searched in the following order :

a) the directory specified using the /I command line option
b) the standard directory set by the environment variable INCLU8.

1.7.4 Macros In Include Directive

Syntax :

***# include** token_sequence*

The above **#include** directive causes the preprocessor to expand the token_sequence.

The expansion of the token sequence must result in one of the following two forms.

a) filename specified within double quotation marks

b) filename specified within angle brackets.

The processing of the **#include** directive depends on the filename specification.

Example 1.28

```
# define FILENAME “file1.c”
# include FILENAME
```

In the above example, the preprocessor includes file1.c.

1.8 CONDITIONAL COMPILATION

1.8.1 Introduction

The following preprocessor directives are used for conditional compilation.

1. # if
2. # ifdef
3. # ifndef
4. # elif
5. # else
6. # endif

These directives allow to suppress compilation of parts of a source file by testing a constant expression or identifier, to determine which parts of the code will be sent to the compiler and which parts of the code will be removed from the source file during preprocessing.

1.8.2 Conditional Compilation Directives

Syntax :

```
#if restricted_constant_expression
        [text-block]
[#elif restricted_constant_expression
        [text-block]
        .
        .
]
[#else
        [text-block]
]
#endif
```

The text-block following the **#if** directive can be any sequence of text. It can occupy more than one line. The text-block may also contain preprocessor directives.

The **#elif** and **#else** directives are optional. Any number of **#elif** directives may appear between **#if** and **#endif** directives. Only one **#else** directive may appear between **#if** and **#endif**. The **#else** directive, if present, must be the last conditional directive before **#endif**. The **#endif** ends the block.

The restricted constant expression in **#if** and subsequent **#elif** are evaluated until an expression with a non-zero value is found. Text following the zero value is discarded. The text following the non-zero value is treated normally. Once a successful **#if** or **#elif** has been found and its text processed, succeeding **#elif** and **#else** lines, together with their text are discarded.

If all the expressions evaluate to zero, and if there is a **#else** directive, the text following the **#else** is processed normally.

Example 1.29

```
# if 1
        function1 () ;
# endif
```

In the above example, the text following **#if** directive is processed.

Example 1.30

```
# if 0
        function1 () ;
# endif
```

In the above example, the text following **#if** directive is discarded as the result of the expression is zero.

Example 1.31

```
# if 1
        function1 () ;
# elif 0
        function2 () ;
# endif
```

In the above example, the text following **#if** directive is processed, since the result of the expression is non-zero. The constant expression following the **#elif** directive is not evaluated. The text following the **#elif** directive is discarded.

Example 1.32

```
# if 0
        function1 () ;
# elif 1
        function2 () ;
# endif
```

In the above example, the text following **#if** will not be processed. The constant expression following **#elif** directive will be evaluated. As the result of the expression is non-zero, the text following the **#elif** directive will be processed.

Example 1.33

```
        # if 0
                function1 () ;
        # elif 0
                function2 () ;
        # endif
```

In the above example, the text following **#if** and **#elif** directives is discarded, as both the expression evaluates to zero.

1.8.3 Restricted Constant Expression

Constant expression in a preprocessor directive is subjected to certain restrictions. The constant expression must be an integral constant expression. It must not contain sizeof expression, enumeration constant, floating point constant and cast expression.

If macros are present they will be expanded. All identifiers remaining after macro expansion are replaced by 0L.

The following illustrates the usage of the restricted constant expression in #if and #elif directives:

Example 1.34

```
        # if 1 +2
```

Example 1.35

```
        # if 1 + 2 * 3 % 4
```

Example 1.36

```
        # if A
```

Example 1.37

```
        # if (1 + 2) / 5
```

Example 1.38

```
        # if (1 << 2) == A
```

Example 1.39

```
        # if A || B & C
```

Example 1.40

```
        # if A ? B : C
```

The following are erroneous:

Example 1.41

```
# if A = 2
```

Example 1.42

```
# if X += 5
```

Example 1.43

```
# if X ++
```

Example 1.44

```
# if &X
```

Example 1.45

```
# if sizeof (struct A)
```

Example 1.46

```
# if A, C
```

Example 1.47

```
# if 1.2
```

1.8.4 defined Operator

Syntax :

defined identifier
defined (identifier)

Any expression of the above syntax is replaced by 1L if the identifier is defined in the preprocessor and by 0L if not.

Example 1.48

```
# define A 1
# if defined (A)
        printf (“This part will be compiled”) ;
# endif

# if defined (B)
        printf (“This part will not be compiled”) ;
#endif
```

The defined operator may also appear with other operators.

Example 1.49

```
        # define A 1
        # if ! defined (A)
                printf ("This part will not be compiled") ;
        # endif

        # if defined (A) - 1
                printf ("This part will not be compiled") ;
        # endif
```

1.8.5 Nesting

The **#if**, **#elif**, **#else** and **#endif** directives may be nested in the text portions of other **#if** directives. Each **#elif**, **#else** and **#endif** directive belongs to the closest preceding **#if** directive.

Example 1.50

```
        # if 0
                # if 1
                        printf ("This part will not be compiled") ;
                # endif
        # endif

        # if 1
                # if 0
                        printf ("This part will not be compiled") ;
                # endif

                # if 1
                        printf ("This part will be compiled") ;
                # endif
        # endif
```

Nesting level is restricted to 32.

1.8.6 Testing Symbol Definition With #ifdef and #ifndef

Syntax :

> *# **ifdef** identifier*
> *# **ifndef** identifier*

The **#ifdef** and **#ifndef** directives may occur wherever a **#if** directive can occur. The text following the **#ifdef** directive is compiled if the specified identifier is a macro. The text following the **#ifndef** directive is compiled when the specified identifier is not a macro.

Example 1.51

```
# define A 1
# ifdef A
        printf (“This part will be compiled”) ;
# endif

# ifdef B
        printf (“This part will not be compiled”) ;
# endif

# define B 2

# ifndef B
        printf (“This part will not be compiled”) ;
# endif

# undef A

# ifndef A
        printf (“This part will be compiled”) ;
# endif
```

1.9 LINE

Syntax :

> *# **line** constant [“filename”]*

The **#line** directive causes the preprocessor to change the following :

a) The number of the next source line to the number specified by the constant

b) The name of the current source file to the specified filename.

The constant value must be a integer constant. This value must be between 1 and 32767, inclusive of both.

The filename is optional. The filename must be enclosed within double quotes as a string literal.

Macros in the **# line** directive are expanded before interpretation.

The line number and the filename are used by the compiler in specifying the error messages during compilation.

Example 1.52

```
# line 124
```

The line number of the next source line is changed to 124. The name of the source file is not changed.

Example 1.53

```
# line 1234 “file.c”
```

The line number of the next source line is changed to 1234. The name of the source file is changed to “file.c”.

Example 1.54

```
# define LINENUMBER 1234
# define FILENAME “file1.c”
# line LINENUMBER FILENAME
```

The line number of the next source line is changed to 1234. The name of the source file is changed to file1.c.

1.10 ERROR

Syntax :

*# **error** [token_sequence]*

The **# error** directive causes the preprocessor to display a diagnostic error message that includes the optional token sequence.

The compiler displays the error message with an error number, the source filename and source line number.

Macros in the token sequence are not expanded.

Example 1.55

```
# error this is an old version
```

The above **# error** directive causes the compiler to display the message "**# error** : this is an old version".

Example 1.56

```
# define ERROR_MESSAGE this is the error message
# error ERROR_MESSAGE
```

The above **#error** directive causes the compiler to display the message "ERROR_MESSAGE". The macro is not expanded.

1.11 MIXED LANGUAGE PROGRAMMING

Syntax :

```
# asm
        [ assembly text ]
# endasm
```

The **#asm** and **#endasm** directives facilitate mixed language programming. The assembly text specified between **#asm** and **#endasm** will not be processed.

The assembly text is not restricted to a single line.

#asm directive marks the beginning of assembly text. The **#endasm** directive marks the end of assembly text.

Example 1.57

```
# asm
l           er0,        NEAR _b                         ;; a = b + c
l           er2,        NEAR _c
add         er0,        er2
st          er0,        NEAR _a
# endasm
```

1.12 PREDEFINED MACROS

The following macros are predefined.

1. __LINE__
2. __FILE__
3. __DATE__
4. __TIME__
5. __STDC__
6. __CCU8__
7. __VERSION__
8. __ARCHITECTURE__
9. __DEBUG__
10. __MS__
11. __ML__
12. __NEAR__
13. __FAR__
14. __UNSIGNEDCHAR __
15. __NOROMWIN__

The above predefined macros cannot be redefined or undefined.

1. __LINE__

   __LINE__ expands to a decimal constant. The decimal constant contains the number of the current source line being compiled.

2. __FILE__

   __FILE__ expands to a string literal. The string literal contains the name of the file being compiled.

3. __DATE__

   __DATE__ expands to a string literal. The string literal contains the date of compilation in the following format.

   “Mmm dd yyyy”

4. __TIME__

   __TIME__ expands to a string literal. The string literal contains the time of compilation in the following format.

   “hh:mm:ss”

5. __STDC__

   __STDC__ expands to a decimal constant 0. The value of the constant is intended to be 1 only in the implementation conforming to ANSI standard.

6. __CCU8__

   __CCU8__ expands to a decimal constant 1.

7. __VERSION__

   __VERSION__ expands to a string literal. The string literal contains the current version number in the following format.

   “Ver.X.YY”

   where X.YY is the current version number.

8. __ARCHITECTURE__

    __ARCHITECTURE__ expands to a string literal. The string literal contains the core specified with /T option in the following format:

    “core”

    where core is the string specified with the /T option. When /T option is not specified then the replacement text will be “”.

9. __DEBUG__

    The above identifier expands to a decimal constant 1, if the ‘C’ source program is compiled with **/SD** option. Otherwise this macro is not defined.

10. __MS__

    The above identifier expands to a decimal constant 1, if the ‘C’ source program is compiled with /MS option or with default memory model option. Otherwise this macro is not defined.

11. __ML__

    The above identifier expands to a decimal constant 1, if the ‘C’ source program is compiled with **/ML** option. Otherwise this macro is not defined.

12. __NEAR__

The above identifier expands to a decimal constant 1, if the ‘C’ source program is compiled with **/near** option or with default data access specifier option or the macro is specified after the pragma **NEAR**. Otherwise this macro is not defined.

13. __FAR__

The above identifier expands to a decimal constant 1, if the ‘C’ source program is compiled with **/far** option or the macro is specified after the pragma **FAR**. Otherwise this macro is not defined.

14. __UNSIGNEDCHAR__

    __UNSIGNEDCHAR__ expands to a decimal constant 1, if the C source program is compiled with /J option. Otherwise this macro is not defined.

15. __NOROMWIN__

__NOROMWIN__ expands to a decimal constant 1, if the C source program is compiled with /NOWIN option. Otherwise this macro is not defined.

Consider the source filename as test.c, the number of the source line being compiled as 200, the date of compilation as 20 September 2000 and the time of compilation as 10 hours : 20 minutes : 30 seconds. The predefined macros expand as follows.

Example 1.58

MACRO CALL	REPLACEMENT TEXT
__LINE__	200
__FILE__	“test.c”
__DATE__	“Sep 20 2000”
__TIME__	“10:20:30”
__STDC__	0

Example 1.59

```
int i ;
void
func (void)
{
        i = __MS__ ;
}
```

If the above C source program is compiled with /MS option or with default memory model option, then it is equivalent to

```
int i ;
void
func (void)
{
        i = 1 ;
}
```

2. LEXICAL CONVENTIONS

2.1 CHARACTER SET

This section describes the lexical conventions adopted by CCU8. After preprocessing, the source program is reduced to a series of tokens based on the lexical conventions.

‘C’ character set consists of the letters, digits and punctuation marks having specific meanings in the ‘C’ language. ‘C’ program is constructed by combining the characters of the ‘C’ character set into meaningful statements.

The following characters can be used in ‘C’ to form constants, identifiers and keywords:

English characters			(A-Z, a-z)						
Numerals			(0 - 9)						
!	#	‘	“	%	&	(	)	=	~
-	^	\	|	,	.	/	?	{	}
<	>	;	:	+	*	[	]	_	
SPACE(20H)			TAB(09H)			CR(0DH)			
LF(0AH)			FF(0CH)			VT(0BH)			

CCU8 treats upper-case and lower-case letters as distinct characters.

Blanks(spaces), horizontal and vertical tabs, new-lines, line- feeds, carriage-returns, and form-feeds are collectively called as white-space characters. Compiler considers them as separators of tokens and ignores. These characters separate user defined items, such as constants and identifiers, from other items in the program.

2.2 TOKENS

In a ‘C’ source program, the basic element recognized by the compiler is the character group known as a “token”. A token is source program text, the compiler will not attempt to further analyze into component elements. The tokens recognized by CCU8 are :

* Identifiers
* Keywords
* Comments
* Constants
* Operators

2.2.1 Identifiers

An identifier is a sequence of letters, digits and underscores. The first character must be a letter or underscore. By default, CCU8 assumes maximum identifier length as 31. If an identifier exceeding this length is specified, CCU8 outputs a warning message and considers only the first 31 characters. However, CCU8 provides a command line option /SL for the user to specify the maximum length of an identifier. User may specify a length ranging from 31 to 254, inclusive of both.

Following are examples of identifiers :

Example 2.1

```
i
count
number
end_of_file
Minus
SUBTRACT_THIS
_var
```

2.2.2 Keywords

Identifiers which are set aside by the compiler for its use are keywords. They cannot be redeclared. They identify data types, storage class and statements in CCU8. Keywords must be expressed in lower-case letters. CCU8 reserves the following words as keywords :

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			

CCU8 supports the following keywords that are extensions from ANSI C standard. However if **/Za** option is specified, these keywords are treated as identifiers.

__DI	__EI	__asm	__far	__near	__noreg

2.2.3 Comments

Comments, delimited by the character pairs (/*) and (*/), can be placed anywhere a white-space can appear. The text of a comment can contain any character except the close comment delimiter (*/). Comments cannot be nested and cannot occur within string or character literal.

Example 2.2

```
i. /* This is a comment */
ii. /* Comments /* nesting */ is not allowed */
```

The second line (ii) would result in error.

CCU8 compiler support an extension for comments. The symbol // can be used at any point in a physical source line (except inside a character constant or string literal). Any characters from the // to the end of the line are treated as comment characters. The comment is terminated by the end of the line.

Example 2.3

```
// close all the files
for (lvar = 0; lvar < lcount; lvar++ )
{
        a++ ;                    // increment a
}
```

2.2.4 Constants

Constants in ‘C’ refer to fixed values, characters and character strings, which cannot be altered by the program. CCU8 supports four types of constants - integral, floating, character and strings.

2.2.4.1 INTEGRAL CONSTANTS

Integer constants represent values themselves in hexadecimal, decimal or octal format. The first character of a decimal integral constant must be a digit. A sequence of digits preceded by 0X or 0x is taken to be hexadecimal integer. If the sequence of digits begin with 0, it is octal; otherwise it is decimal integral constant.

	Valid Characters	**Prefix**
Hexadecimal	0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f	0X or 0x
Decimal	0 1 2 3 4 5 6 7 8 9	None
Octal	0 1 2 3 4 5 6 7	0

An integral constant may be suffixed by the letter ‘u’ or ‘U’ to specify that it is unsigned. It can also be suffixed by ‘l’ or ‘L’ to specify that it is long.

Every integral constant is given a type based on its value. The type of constant determines the conversion to be performed on it when is used in an expression. Conversion rules are summarized below :

∗ The type of an integer constant depends on its form, value and suffix. The type of an integer constant is the first of the corresponding list in which its value can be represented.

Unsuffixed decimal	: **int, long int, unsigned long int**
Unsuffixed octal or hexadecimal	: **int, unsigned int, long int, unsigned long int**
Suffixed by the letter **u** or **U**	: **unsigned int, unsigned long int**
Suffixed by the letter **l** or **L**	: **long int, unsigned long int**
Suffixed by both the letters **u** or **U** and **l** or **L**	: **unsigned long int**

The following table shows the range of values and the corresponding type for octal and hexadecimal constants in CCU8 where int type is 16 bits long.

Hexadecimal range	**Octal range**	**Type**
0x0 to 0x7fff	0 to 077777	int
0x8000 to 0xffff	0100000 to 0177777	unsigned int
0x10000 to 0x7ffffff	0200000 to 017777777777	long
0x80000000 to 0xffffffff	020000000000 to 037777777777	unsigned long

An integer constant can be forced to long type by appending the letter ‘l’ or ‘L’.

Some examples of integer constants are shown below:

Example 2.4

```
0x177AF      /* Hexadecimal integer */
0167         /* Octal integer */
1826         /* Decimal integer */
0X1abe       /* Hexadecimal integer */
10l          /* Decimal long integer */
0xabL        /* Hexadecimal long integer */
0333l        /* Octal long integer */
```

2.2.4.2 FLOATING-POINT CONSTANTS

A floating-point constant has an integral part (decimal part), a fractional part (the letter e or E), and an optionally signed integer exponent. The integral and fractional parts consist of decimal digits; one of which can be omitted. Omission of either decimal point with the following digits or the E (exponent) is allowed, but both cannot be omitted.

Floating-point constants may be of type **float** or **double.** The type is determined by the suffix; F makes it float, L or l makes it **long double**; otherwise it is **double. Long double** constants are treated similar to **double** constants. The following are examples of floating-point constants:

Example 2.5

```
1.0e10f
.75
1.03e-12L
3.0
120e22
10e04
-0.0021
```

2.2.4.3 CHARACTER CONSTANTS

Character constants are formed by a single ASCII character enclosed within single quotation marks (‘’). Only one byte characters can be used for character constants. An escape sequence is regarded as a single character and is therefore valid in a character constant. To use a single quotation mark or backslash character as a constant, a backslash must precede them.

Example 2.6

```
‘ ’ Single blank space
‘z’ Lower-case z
‘\n’ Newline character
‘\\’ Backslash
‘\’’ Single quote
```

2.2.4.4 STRING CONSTANTS

A string constant also called a string literal, is a sequence of characters surrounded by double quotes (“..”). A string has type “array of characters” and storage class **static** and is initialized with the given characters.

Adjacent string literals are concatenated into a single string. After concatenation, a null byte **‘\0’** is appended to the string so that programs that scan the string can find its end. All strings even if not concatenated are appended with a null byte in order to indicate its end. String literals can contain escape sequences.

To form a string literal that takes up more than one line is to type a backslash and then to press the RETURN key. The backslash causes the compiler to ignore the new-line character immediately following the backslash. For example,

```
“This string in two lines is combined \
into a single line string.”
```

is same as

```
“This string in two lines is combined into a single line string.”
```

Two or more strings separated only by white space characters are concatenated into a single string. For example, the following strings :

```
“This is first,”
“ this is second.”
```

will be concatenated as

```
“This is first, this is second.”
```

Escape sequences can be used in a string literal. To use double quotation mark or backslash within a string literal, escape sequences should be used.

Example 2.7

i. “One\\two”

ii. “\”Do it\” Mike said.”

2.2.4.5 ESCAPE SEQUENCES

Strings and character constants can contain “escape sequences”. Escape sequences are character combinations representing whitespace and non-graphic characters. An escape sequence consists of a backslash (\) followed by a letter or by a combination of digits.

Escape sequences are typically used to specify actions such as carriage returns and tab movements on terminals and printers and to provide literal representations of non-printable characters and characters that normally have special meanings, such as the double quotation mark character (“). The following table lists the CCU8 escape sequences.

Escape sequence	**Name**	
\n	New line	NL (LF)
\t	Horizontal tab	HT
\v	Vertical tab	VT
\b	Backspace	BS
\r	Carriage return	CR
\f	Formfeed	FF
\a	Bell (alarm)	BEL
\’	Single quote	
\”	Double quote	
\\	Backslash	
\ooo	ASCII character in octal notation	
\xhh	ASCII character in hexadecimal notation	

If a backslash precedes a character that does not appear in the above table, the backslash is ignored and the character is represented literally. For example, the pattern “\m” represents the character “m” in a string literal or character constant.

The sequence \ooo allows the programmer to specify any character in the ASCII character set as a three-digit octal character code. The hexadecimal digits that follow the backslash (\) and the letter **x** in a hexadecimal escape sequence are taken to be part of the construction of a single character for an integer character constant. The numerical value of the hexadecimal integer so formed specifies the value of the desired character. Each hexadecimal escape sequence is the longest sequence of characters that constitute the escape sequence. For example the ASCII horizontal tab character can be given as the normal ‘C’ escape sequence \t or can be coded as \011 (octal) or \x09 (hexadecimal).

Atleast one digit must be specified for both octal and hexadecimal escape sequence. For example \11, \011, \x9 and \x09 are valid escape sequences.

2.2.5 Operators

Operators are symbols that specify how values are to be manipulated. Each symbol is interpreted as a single unit called a “token”. The following tables list ‘C’ unary, binary and ternary operators.

UNARY OPERATORS

!	~	-	*	&	+	++	--	sizeof

BINARY OPERATORS

+	-	*	/	%	<<	>>
<	<=	>	>=	==	!=	&
|	^	&&	||	,	=	+=
-=	*=	/=	%=	>>=	<<=	&=
|=	^=					

TERNARY OPERATOR

?:

Four operators *****, **&**, **-** and **+** appear in both unary and binary tables shown above. Their interpretation as unary or binary depends on the context in which they appear.

3. PROGRAM STRUCTURE

3.1 SOURCE PROGRAM

This section defines the terms that are used later in the manual to describe the ‘C’ language as implemented by CCU8 and discusses the structure of ‘C’ programs.

A ‘C’ source program is a collection of any number of directives, declarations, definitions and statements. These constructs are described briefly below. These constructs can appear in any order in a program.

DIRECTIVES

A directive instructs the ‘C’ preprocessor to perform a specific action on the text of the program before compilation. Directives are described in section dealing with PREPROCESSOR.

DECLARATIONS AND DEFINITIONS

A declaration establishes an association between the name and the attribute of a variable, function or type. In ‘C’, all variables must be declared before being used.

A definition of a variable establishes same associations as a declaration, but also causes storage to be allocated for the variable. All definitions are declarations but not all declarations are definitions.

Example : 3.1

```
const int a = 10 ;                  /* Variable definitions */
int b ;                             /* at external level */
extern int function (int, char) ;   /* Function declaration or prototype */
extern long c ;                     /* Variable declaration at external level     */
extern float f ;

main ()
{
        int local1 ;                /* Variable definitions at */
        char local2 ;               /* internal level */

        local1 = local2 ;           /* Executable statements */
        c = b + a + f ;
}
```

3.2 SOURCE FILES

A source program can be divided into one or more source files. A ‘C’ source file is a text file containing all or part of a ‘C’ program. During compilation individual source files must be compiled separately.

A source file can contain any combination of directives, declarations and definitions. Items such as function definitions or large data structures cannot be split between source files. The last character in a source file must be new-line character or end of file.

A source file need not contain executable statements. For example, it may be useful to place definitions of variables in one source file and then declare references to these variables in other source files that use them. This technique make definitions easy to find and change. For the same reason macros and **#define** statements are often organized into separate include files that may be referenced in source files as required.

Directives in a source apply only to that source file and its include files. Moreover, each directive applies only to the part of the file that follows the directive. To apply a common set of directives to a whole source program the directives must be included in all source files comprising the program.

3.3 FUNCTIONS AND PROGRAM EXECUTION

Every ‘C’ program has a primary (main) function that must be named main. The main function serves as the starting point for program execution. It usually controls program execution by directing the calls to other functions in the program. A program usually stops executing at the end of main, although it can terminate at other points in the program for a variety of reasons depending on the execution environment.

The source program usually has more than one function, with each function designed to perform one or more specific tasks. The main function calls these functions to perform one or more specific tasks. When main function calls another function, it passes execution control to that function, so that execution begins at the first statement in the called function. This function returns control when a return statement is executed or when the end of the function is reached.

Functions can be declared to have parameters. When such a function calls another, the called function receives values from the calling function. These values are called arguments.

Arguments are passed between functions using call by value method.

3.4 LIFETIME AND VISIBILITY

Three concepts are crucial to understanding the rules that determine how variables and functions can be used in a program. They are blocks (or compound statement), lifetime (sometimes called extent) and visibility (sometimes called scope).

3.4.1 Blocks

A block is a sequence of declarations, definitions and statements enclosed within curly braces. There are two types of blocks in ‘C’. The compound statement is one type of block. The other, the function definition, consists of a compound statement comprising the function body plus the header associated with the function (the function name, return type and formal parameters). A block may encompass other blocks, with the exception that no block can contain a function definition. A block within other blocks is said to be nested within the encompassing blocks.

All compound statements are enclosed in curly braces. However everything enclosed within curly braces do not constitute a compound statement. For example, though the specification of array or structure elements may appear within curly braces, they are not considered compound statements.

3.4.2 Lifetime

Lifetime is the period, during execution of a program, in which a variable or function exists. All functions in a program exist at all times during its execution.

Lifetime of a variable may be global or local. If its lifetime is global (a global item), it has storage and a defined value for the entire duration of a program. An item with a local lifetime has storage and a defined value only within a block where the item is defined or declared. A local item is allocated new storage each time program enters that block and it loses its storage (and hence its value) when the program exits the block.

3.4.3 Visibility

Visibility determines the portions of the program in which an item can be referenced by name. An item is visible only in portions of a program encompassed by its scope which may be restricted to the file, function, block or function prototype in which it appears.

3.5 NAMING CLASSES

In any ‘C’ program identifiers are used to refer to many different kinds of items. Identifiers have to be provided for functions, variables, formal parameters, union members and other items the program uses. ‘C’ allows to use the same identifier for more than one program item, as long as the rules outlined in this section are followed.

The Compiler sets up naming classes to distinguish between the identifiers used for different kinds of items. The names within each class must be unique to avoid conflict, but an identical name can appear in more than one naming class. This means that the same identifier can be used for two or more items provided that the items are in different naming classes. The compiler resolves the references based on the context of the identifier in the program.

The following list describes the kinds of items that can be named in ‘C’ program and the rules for naming them :

Statement labels

Statement labels form a separate naming class. Each statement label must be distinct from all other statement labels in the same function. Statement labels do not have to be distinct from other names or label names in other functions.

Variables and Functions

The names of variables and functions are in a naming class with formal parameters and typedef names. Therefore, variables and function names must be distinct from other names in this class that have the same visibility. However, variables and function names can be redefined within function blocks.

Formal parameters

The names of formal parameters to a function are grouped with the names of the local variables, so the formal parameter names should be distinct from the local variable names. The formal parameters cannot be redefined at the top level of the function. However the names of the formal parameters may be redefined in subsequent blocks nested within the function body.

typedef names

The names of types defined with the '**typedef**' keyword are in a naming class with variable and function names. Therefore, **typedef** names must be distinct from all variable and function names with the same visibility as well as from the names of formal parameters. Like variable names, names used for **typedef** types can be redefined within program blocks.

Tags

Structure, union and enum tags are grouped in a single naming class. These tags must be distinct from other tags with the same visibility. Tags do not conflict with any other names.

Members

The members of each structure and union form a naming class. The name of a member must, therefore, be unique within the structure or union, but it does not have to be distinct from other names in the program, including the names of members of different structures and unions.

Example 3.2

```
struct name {
                char * name ;
                int type ;
                int scope ;
            } name ;
```

Since structure tags, structure members and variable names are in three different naming classes, the three items named “name” in this example do not conflict and are distinct.

3.6 DATA TYPES

CCU8 supports several basic data types and derived data types.

BASIC TYPES

There are several fundamental types supported by CCU8. They include **char**, **int**, **long**, **float** and **double**. Three sizes of integers are available namely, **short int**, **int**, and **long int**. Both **signed** and **unsigned** objects of **char** and **int** types can be declared. The size as well as the smallest and largest values of each type are mentioned in section 4.2.

Objects of all the above mentioned basic types can be interpreted as numbers. Therefore they will be referred to as arithmetic types.

Types **char** and **int** of all sizes, each with or without sign will collectively be called as integral types.

The types **float**, **double** and **long double** will be called as floating point type.

DERIVED TYPES

Besides basic types, there is conceptually infinite class of derived types constructed from the fundamental types in the following ways :

Arrays of objects of a given type.

Functions returning objects of a given type.

Pointers to objects of a given type.

Structures containing a sequence of objects of various types.

Unions capable of containing any one of several objects of various types.

4. DECLARATIONS

4.1 INTRODUCTION

Declarations specify the interpretation given to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations that reserve storage are called definitions. Declarations have the form

declaration : [declaration_specifiers] [init_declarator_list] ;

declaration : __asm (string)

All ‘C’ variables must be explicitly declared before being used.

Declarators contain the identifiers being declared that may be modified with brackets ([]), asterisks (*) or parentheses. Declaration specifiers consist of a sequence of type and storage class specifiers.

```
declaration_specifiers :
        storage_class_specifier [declaration_specifiers]
        type_specifiers [declaration_specifiers]
        type_qualifiers [declaration_specifiers]

init_declarator_list :
        init_declarator
        init_declarator_list , init_declarator

init_declarator :
        declarator
        declarator = initializer
```

4.2 TYPE SPECIFIERS

The type specifiers supported by CCU8 are listed below :

void	char	int	short	enum
long	float	double	signed	
unsigned	struct	union	typedef	

The keywords **signed** and **unsigned** can precede any of the integral types and can also be used alone as type specifiers, in which case they are understood as **signed int** and **unsigned int** respectively.

When used alone the keyword **int** is assumed to be **signed int**. When used alone, the keywords **long** and **short** are understood as **long int** and **short int** respectively. By default if only **char** is specified, it is treated as **signed char**. However, if **/J** option is specified in the command line, default **char** is treated as **unsigned char** by the compiler.

The **signed char**, **signed int**, **signed short int** and **signed long int** types, together with their unsigned counterparts are called integral types. The **float** and **double** type specifiers are referred to as floating-point type. Any of these integral and floating-point type specifiers can be used in a variable or function declaration.

The keyword **void** has three uses :

1. **void** is used to declare a function that returns no value.
2. To declare a pointer to an unspecified type.
3. When **void** occurs alone within the parentheses following the function name, **void** indicates that the function accepts no arguments.

The storage and range of values for fundamental type are summarized below :

Type	**Storage**	**Range of values**
Char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255
short,int	2 bytes	-32,768 to 32,767
unsigned short,unsigned int	2 bytes	0 to 65,535
Long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295
Float	4 bytes	IEEE-standard notation 1.17549435e-38 to 3.40282347e+38
Double	8 bytes	IEEE-standard notation 2.2250738585072014e-308 to 1.7976931348623157e+308

The **long double** type specifier may also be used. It is treated similar to **double** type specifier.

4.3 TYPE QUALIFIERS

1. **const**
2. **volatile**

Types may also be qualified, to indicate special properties of the objects being declared. The type qualifiers supported by CCU8 are **const** and **volatile.**

The **const** type qualifier is used to declare an object as non-modifiable. The **const** keyword can be used as a qualifier for any fundamental or aggregate type. A **typedef** may be qualified by a **const** type qualifier. A declaration that includes the keyword **const** as a qualifier of an aggregate type declarator indicates that each element of the aggregate type is not modifiable. If an item is declared with only the **const** type qualifier, its type is taken to be **const int**.

CCU8 allocates such variables in Read Only Memory (ROM). The **const** type qualifier may be used only with global variables. If **/NOWIN** option is not specified in the command line, then CCU8 allocates **const** qualified near variables in the ROMWINDOW region. If **/NOWIN** option is specified in the command line, then CCU8 issues warning message for **const** qualified near variables as it cannot be allocated in the ROMWINDOW region.

CCU8 ignores **const** qualifier for local automatic variables and function parameters, after issuing a warning message. If **/Za** option is specified, function parameters may be qualified with **const** and if these parameters are modified, error is issued.

Individual members of a structure cannot be qualified by **const**. However, if **/Za** option is specified, members may be qualified as **const** and if they are modified, error is issued.

In case of structure and union, tags cannot be qualified by **const**. Only **struct/union** variables can be qualified by **const**. CCU8 ignores **const** in case of **structure** and **union** tags. The **const** qualifier along with the struct/union tags are taken as qualifier for the variables, if any, specified along with the **struct/union** tag declaration.

Example 4.1

```
const struct tag {
                int a ;
                char b ;
            } var0 ;

struct tag var1 ;
const struct tag var2 ;
```

In the above example although const is used in the declaration of the **struct** tag ‘tag’, it is ignored. Thus variables declared using this ‘tag’ must be qualified by **const** in order to reside in read only memory, however, variables defined with the **struct** tag ‘tag’ declaration are qualified by **const**. Thus, in the above example, ‘var1’ is not qualified by **const**, but ‘var0’ and ‘var2’ are qualified by **const**.

Individual members of a structure cannot be qualified by const.

Example 4.2

```
struct tag1 {
            int a ;
            const char b ;
        } var1 ;

struct tag1 var2 ;
```

In the above example although **const** is used in the declaration of the structure member ‘b’, it is ignored after issuing a warning.

A typedef identifier may be qualified by **const**.

Example 4.3

```
typedef const int ca ;
ca const_identifier ;
```

In the above example the typedefed identifier is qualified by **const**. Hence the variable ‘const_identifier’ declared using the typedefed variable ‘ca’ is also qualified by **const**.

The **volatile** type qualifier declares an item whose value may legitimately be changed by something beyond the control of the program in which it appears.

The **volatile** keyword can be used in the same circumstances as const. An item may be both const and volatile.

Items qualified by volatile will suppress optimization of expressions in which they are used.

Example 4.4

```
volatile int input ;
volatile char * key_ptr ;
```

In the above example, value of ‘input’ may change beyond the control of program. Similarly, the contents of location pointed to by ‘key_ptr’ may change beyond the control of program.

4.4 DECLARATORS

```
declarator :
        [pointer] direct_declarator

direct_declarator :
        [memory_function_qualifier_list] identifier
        ( declarator )
        direct_declarator [constant_expression]
        direct_declarator (parameter_type_list)
        direct_declarator ([identifier_list])

pointer :
        [memory_function_qualifier_list]* [type_qualifier_list]
        [memory_function_qualifier_list]* [type_qualifier_list] pointer

type_qualifier_list :
        type_qualifier
        type_qualifier_list type_qualifier

memory_function_qualifier_list :
        function_qualifier
        memory_function_qualifier_list  memory_model_qualifier
        memory_model_qualifier
        memory_function_qualifier_list function_qualifier
```

‘C’ language allows a programmer to declare arrays of values, pointers to values and functions returning values of specified types. A declarator must be used to declare these items.

A declarator is an identifier that may be modified by brackets ([]), asterisks (*) or parentheses (()) to declare an array, pointer or function type respectively. Declarators appear in the pointer, array and function declarations.

When a declarator consists of an unmodified identifier, the item being declared has a basic type. If asterisks appear to the left of an identifier, the type is modified to a pointer type. If the identifier is followed by brackets ([]), the type is modified to an array type. If the identifier is followed by parenthesis, the type is modified to a function type.

Example 4.5

```
i.   int table [100] ;
ii.  char * cp ;
iii. long function (void) ;
```

In the above example, (i) declares an array of integers, named table, containing 100 values. (ii) declares a pointer to a character value, cp. (iii) declares a function that returns a long value and takes no arguments.

A ‘complex’ declarator is an identifier modified by more than one array, pointer or function modifier. Various combinations of array, pointer, and function modifiers can be applied to a single identifier. However, a declarator may not have the following illegal combinations :

1. An array cannot have function as its elements.
2. A function cannot return an array or a function.

Example 4.6

```
i.   int (* ((* fnarray []) ())) () ;    /* correct */
ii.  int ((* func []) ()) [] ;           /* error */
```

In the above example, (ii) is an error because it specifies an array of pointers to functions returning an array of integers.

For declarations within global scope, the declarations specifier is optional. If no declaration specifier is specified in a declaration, CCU8 assumes **int** as the default declaration specifier. However, if **/Za** option is specified for such declarations and for empty declarations, error is issued.

4.4.1 Memory Model Qualifiers

Memory model qualifiers can be used in a declaration, to explicitly specify the addressing type of the variable. **__near and __far** are the three keywords supported by CCU8 that can be used to specify the addressing type of an object. A memory model specifier affects the token immediately to it’s right. Memory model qualifiers can qualify only objects and pointers to object.

Example 4.7

```
int * __far fvar ;       /* ‘fvar’ need not be in default segment, but points to an object of
                            type int in default segment. */

int __far * fptr ;       /* ‘fptr’ is in default segment, pointing to an object of type int that
                            need not be in default segment */

__far int evar ;         /* error, as __far cannot qualify a type specifier */
```

__near/__far keyword is used to qualify data. If the object is qualified by **const** and **__near**, storage will be allocated for the object in the ROMWINDOW region. Similarly, if a non-const object is qualified by **__near**, storage will be allocated for the object in default data segment(#0). If the object is qualified by **const** and **__far**, storage will be allocated for the object in any one of the table segment in Data Memory segments. Similarly, if a non-const object is qualified by **__far**, storage will be allocated for the object in any one of the data segment in Data Memory segments.

The __near and __far specifiers cannot be used with functions.
All these memory model qualifiers are mutually exclusive.

Structure and union**,** tags cannot be qualified by **__near/__far**. Only **struct/union** variables can be qualified by **__near/ __far**. CCU8 issues error if struct/union tags are qualified with **__near/far**.

Example 4.8

```
struct __far tag1 {                /* error */
                      int a ;
                      char b ;
                   } var0 ;

struct tag {
                int a ;
                char b ;
           } var0 ;

struct tag1 var1 ;
struct tag2 __far var2 ;           /* var2 is far structure */
```

In the above example, CCU8 issues error for **struct** ‘tag1’ declaration, as the **struct** tag is qualified with **__far**. The variable ‘var2’ is qualified with **__far** and therefore it is allocated in any one of the Data Memory segments.

Individual members of a structure cannot be qualified by **__near**/**__far**.

Example 4.9

```
struct tag1 {
```

```
                    int           a ;
                    char __far  b ;
                } var1 ;

        struct tag1 var2 ;
```

In the above example although **__far** is used in the declaration of the structure member ‘b’, it is ignored after issuing a warning.

A typedef identifier may be qualified by **__near and __far**.

Example 4.10

```
        typedef int __far FVAR ;
        FVAR far_identifier ;
```

In the above example the typedefed identifier ‘FVAR’ is qualified by **__far**. Hence the variable ‘far_identifier’ declared using the typedef name ‘FVAR’ is also qualified by **__far**.

4.4.2 Function Qualifiers

CCU8 supports **__noreg** keyword that can qualify functions only. If **__noreg** is used to qualify any other object, error is issued. When a function is qualified with **__noreg,** CCU8 passes arguments by stack and not by register. If a function is qualified with more than one **__noreg**, error is issued.

Example 4.11

```
        int __noreg fn1 ( int arg1 ) ;      /*  value of arg1 will be passed through stack and not
                                                though register */
        int fn2 ( int arg2 ) ;              /*  value of arg2 will be  passed through register */


        int __noreg var ;                   /*  error : as var is not a function */
```

Example 4.12

```
        int gint ;
        int __noreg fn3 (int arg3)          /* function definition qualified by __noreg */
                                            /* function will take arguments from stack */
            gint = arg1 ;
        }
```

4.4.3 Interpreting Declarations

‘C’ programming language syntax for declaring objects is unlike the declaration syntax of other languages. The exact meaning of a complex ‘C’ declaration is not always immediately apparent. A complex declarator is an identifier qualified by more than one array, pointer or function modifier.

In interpreting complex declarators, brackets and parentheses (that is modifiers to the right of the identifier) take precedence over asterisks (that is modifiers to the left of the identifier).

Brackets and parentheses have same precedence and associate from left to right. After the declarator is fully interpreted, the type specifier is applied as the last step. By using parentheses default association order can be overridden and a particular interpretation can be forced.

A simple way to interpret complex declarators is to read them from inside out using the following steps :

1. Start with the identifier and look to the right for brackets or parentheses (if any).
2. Interpret these brackets or parentheses, then look to the left for asterisks.
3. If a right parenthesis is encountered at any stage, go back and apply rules 1 and 2 to everything with in the parentheses.
4. Finally apply the type specifier.

Example 4.13

```
char *  ( * ( * cpvar)())[20] ;
^    ^  ^ ^ ^ ^ ^
7    6  4 2 1 3 5
```

In the example the steps are labeled and can be interpreted as follows :

1. The identifier cpvar is declared as
2. a pointer to
3. a function returning
4. a pointer to
5. an array of 20 elements, which are
6. pointers to
7. char values.

Array of pointers to **int** values may be declared as shown below.

Example 4.14

```
int * variable [5] ;
```

The following example shows how a pointer to array of **int** values is declared.

Example 4.15

```
int (* var) [5] ;
```

Example 4.16

```
char *fn (int, char ) ;
```

In example 4.16, a declaration to a function returning a pointer to a **char**, and which takes two arguments as **int** and a **char** is specified.

A declaration for a pointer to function returning a **float** and taking no argument is given below.

Example 4.17

```
float (*fn1)(void) ;
```

4.5 VARIABLE DECLARATIONS

This section describes the form and meaning of variable declarations.

Syntax :

[sc-specifier] type-specifier declarator[,declarator]

In particular, this section explains how to declare the following :

Simple variables	: Single value variables with integral floating-point type
Structures	: Variables composed of a collection of values that may have different types
Unions	: Variables composed of several values of different types, which occupy the same storage space
Arrays	: Variables composed of a collection of elements with the same type
Pointers	: Variables that point to other variables and contain variable locations (in the form of addresses) instead of values.

4.5.1 Simple Variable Declarations

Syntax :

[sc-specifier] type-specifier declarator [,declarator]

The declaration of a simple variable specifies the variable name and type. It can also specify the storage class of the variable. The identifier in the declaration is the name of a variable. The type-specifier is the name of a defined data type.

A list of identifiers separated by comma can be listed to specify several variables in the same declaration. Each identifier in the list names a variable. All variables defined in the declaration have the same type.

Example 4.18

```
int x ;                          /* declares a simple integer variable */

unsigned long int lvar1, lvar2 ; /*  two variables unsigned long int type is declared */

const int init = -1 ;            /* declares an int variable, qualified by const and
                                    initialized to -1 */

int __far fvar ;                 /* declares fvar as int variable that is located in far data
                                    memory */

int __far fvar, nvar ;           /* declares fvar as int variable that is located in far data
                                    memory , but nvar is in near memory*/
```

4.5.2 Structure Declarations

Syntax :

struct [tag] {member-declaration-list} [declarator [,declarator]...];
struct tag [declarator[,declarator]...];

A structure declaration names a structure variable and specifies a sequence of variable values (called members of the structure) that can have different types. A variable of that structure type holds the entire sequence defined by that type.

Structure declarations begin with the struct keyword and have two forms :

∗ In the first form, a member-declaration-list specifies the types and names of the structure members. The optional tag is an identifier that names the structure type defined by member-declaration-list.

∗ The second form uses a previously defined structure tag to refer to a structure type defined elsewhere. Thus member-declaration-list is not needed as long as the definition is visible. Declarations of pointers to structures and typedefs for structure types can use the structure tag before the structure type is defined. However, the structure definition must be encountered prior to any actual use of the structure members, typedef or pointer.

In both forms, each declarator specifies a structure variable. A declarator may also modify the type of the variable to a pointer to the structure type, an array of structures or a function returning a pointer to the structure type. If tag is given, but declarator does not appear, the declaration constitutes a type declaration for a structure tag.

Structure tags must be distinct from other structure / union / enum tags with the same visibility.

A member-declaration-list argument contains one or more variable or bit-field declarations.

Each variable declared in the member-declaration-list is defined as a member of the structure type. Variable declarations within the member-declaration-list have the same form as simple variable declarations, except that the declarations cannot contain storage class specifiers or initializers. The member can have any variable type :basic, array, pointer, structure or union.

A member cannot be declared to have the type of the structure in which it appears. However, a member can be declared as a pointer to the structure type in which it appears as long as the structure type has a tag. This facilitates the creation of linked lists of structures.

A bit-field declaration has the following form :

type-specifier [identifier] : constant-expression;

The constant-expression specifies the number of bits in the bit-field. The type specifier may be **unsigned char** or **unsigned int**. If the type specified is **signed char**, **signed int**, **int** or **char**, CCU8 issues warning message and treats them as unsigned. However, if **/J** option is specified no warning message is issued for **char** specified bit fields, as it is treated as **unsigned char**. If **/Za** option is specified, **char** bit fields are not supported.

Constant-expression must be a non-negative integer value which takes values from 0 to 8 for **unsigned char** members and 0 to 16 for **unsigned int** members. Array of bit-fields, pointers to bit-fields and functions returning bit-fields are not allowed. The optional identifier names the bit-field. Named bit-fields cannot have bit-width of 0. Unnamed bit-fields can be used as dummy fields for alignment purposes. An unnamed bit-field whose width is specified as 0 guarantees that storage for the member following it in the member-declaration-list begins on an integral boundary.

Each identifier in a member-declaration-list must be unique within the list. However, they do not have to be distinct from ordinary variable names or from identifiers in other member-declaration lists.

Storage

Structure members are stored sequentially in the order in which they are declared: the first member has the lowest memory address and the last member the highest. Storage for each member begins on a memory boundary appropriate to its type. Therefore, unnamed spaces (holes) may appear between the structure members in memory.

Sequence of bits are packed as tightly as possible. Consecutive bit-field members of type **char** are stored in the same byte location, as long as their cumulative size is within character size. Similarly, consecutive bit-field members of type **int** are stored in the same word location, as long as their cumulative size is within integer size. If the total size exceeds character size for consecutive char bit field members, as a result of a new bit-field member, a new character is allocated for the new bit-field member. Similarly, if the total size exceeds integer size for consecutive integer bit field members, as a result of a new bit-field member, a new integer is allocated for the new bit-field member. If a bit field member of type **char** is followed by another bit field member of type **int**, or a bit field member of type **int** is followed by another bit field member of type **char**, storage for the new member starts from the next even address boundary.

Example 4.19

i.
```
struct inv_type {
                   char item [40] ;
                   float cost ;
                   float retail ;
                   int item_on_hand ;
                   int lead_time ;
                } inv_vara, inv_varb, inv_varc ;
```

This declares a structure type called inv_type and declares variables inv_vara, inv_varb, inv_varc.

ii.
```
struct inv_type invntry[100] ;
```

The above examples declares a 100 element array of structures of type inv_type.

iii.
```
struct symbol_table {
                   char *name ;
                   int type ;
                   unsigned int scope : 2 ;
                   unsigned int sign : 1 ;
                   unsigned int qualy : 1 ;
                   struct symbol_table * next ;
                } *global ;
```

The above example declares a pointer to a structure of type symbol_table. The structure has a pointer to itself. It has three bit-fields and two other members.

```
iv.   struct {
                unsigned char char_bit1 : 2 ;
                unsigned char char_bit2 : 4 ;
                unsigned int    int_bit1  : 8 ;
                unsigned int    int_bit2  : 1 ;

            }bit_field_str ;
```

The above example declares a structure that has both **char** bit fields and **int** bit fields.

4.5.3 Union Declarations

Syntax :

union [tag] {member-declaration-list} [declarator [,declarator]...];
union tag [declarator[,declarator]...];

A union declaration names a union variable and specifies variable values, called members of the union, that can have different types. A variable with union type stores one of the values defined by that type.

Union declarations have the same form as structure declarations, except that they begin with the **union** keyword instead of the **struct** keyword. The same rules govern structure and union declarations.

Storage

The storage associated with a union variable is the storage required for the largest member of the union. When a smaller member is stored, the union variable may contain unused memory space. All members are stored in the same memory space and start at the same address. The stored value is overwritten each time a value is assigned to a different member.

All members in the union are aligned with the lower memory address of the storage allocated.

Example 4.20

```
union union_type {
                    int intvar ;
                    char charvar ;
                  } union_var ;
```

The above defines an union with union_type, and declares a variable union_var, that has two members intvar and charvar.

The maximum number of levels to which structures or unions may be nested is restricted to 16.

4.5.4 Enumeration Declarations

Syntax :

```
enum [tag] {enum-list} [declarator [, declarator]...] ;
enum tag [declarator [,declarator]...] ;
```

An enumeration declaration gives the name of an enumeration variable and defines a set of named integer constants (the enumeration set). A variable with enumeration type stores one of the values of the enumeration set defined by that type. The integer constants of the enumeration set have int type.

Variables of **enum** type are treated as if they are of type int. They may be used in indexing expressions and as operands of all arithmetic and relational operators.

Enumeration declarations begin with the **enum** keyword, have the two forms shown at the beginning of this section. This is described below:

∗ In the first form, enum-list specifies the values and names of the enumeration set. (The enum-list is described in detail below.) The optional tag is an identifier that names the enumeration type defined by enum-list. The declarator names the enumeration variable. Zero or more enumeration variables may be specified in a single enumeration declaration.

∗ The second form of the enumeration declaration uses a previously defined enumeration tag to refer to an enumeration type defined elsewhere. The tag must refer to a defined enumeration type, and that enumeration type must be currently visible. Since the enumeration type is defined elsewhere, enum-list does not appear in this type of declaration.

In both forms of declaration, if a tag argument is given, but no declarator is given, then it constitutes a declaration for an enumeration tag.

An enum-list has the following form:

```
identifier [= constant-expression]
[, identifier [= constant-expression] ... ]
```

Each identifier in an enumeration list names a value of the enumeration set. By default, the first identifier is associated with the value 0, the next identifier is associated with value 1, and so on through the last identifier in the declaration. The name of an enumeration constant is equivalent to its value.

The optional phrase = constant-expression overrides the default sequence of values. Thus, if identifier = constant-expression appears in enum-list, the identifier is associated with the value given by constant-expression. The constant-expression must have int type and can be negative. The next identifier in the list is associated with the value of constant-expression + 1, unless it is explicitly associated with another value.

The following rules apply to the members of an enumeration set :

- Two or more identifiers in an enumeration set can be associated with the same value.
- The identifiers in the enumeration list must be distinct from other identifiers with the same visibility, including ordinary variable names and identifiers in other enumeration lists.
- Enumeration tags must be distinct from other enumeration, structure, and union tags with the same visibility.

Example 4.21

```
enum levels_tag
{
    start,          /* value = 0 */
    primary,        /* value = 1 */
    secondary,      /* value = 2 */
    final           /* value = 3 */

} levels ;
```

This example defines an enumeration type named levels_tag and declares a variable named levels with that enumeration type. The values associated with identifiers are shown in comments.

Example 4.22

```
enum constants
{
    very_low,       /* value = 0   */
    low = 10,       /* value = 10  */
    medium,         /* value = 11  */
    high = 20,      /* value = 20  */
    very_high       /* value = 21  */
} ;

const enum constants speed = high ;
```

In this example, a value from the set named constants is assigned to a variable named speed. Since the constants enumeration type has already been declared, only the enumeration tag is necessary.

4.5.5 Array Declarations

Syntax :

type-specifier declarator [constant-expression] ;
type-specifier declarator [] ;

An array declaration names the array and specifies the types of its element. It may also define the number of elements in the array. A variable with array type is considered a pointer to the type of the array elements.

Array declarations have two forms as shown in the syntax.

∗ In the first form, the constant-expression argument within the brackets specifies the number of elements in the array. Each element has the type given by type-specifier, which can be any type except void.
∗ The second form omits the constant-expression argument in brackets. This form can be used only if the array is initialized, or declared as a formal parameter, or declared as a reference to an array explicitly defined elsewhere in the program.

In both forms, declarator names the variable and may modify the type of a variable. The brackets ([]) following declarator modify the declarator to array type.

A multidimensional array can be declared by following the declarator with a list of bracketed constant expressions as shown below :

type-specifier declarator[constant-expression] [constant-expression]

Each constant-expression in brackets defines the number of elements in a given dimension. In case of multidimensional array the first constant-expression can be omitted if it is initialized or if it is declared as a formal parameter or if it is a reference to an array explicitly defined elsewhere in the program. If the value of the constant expression is zero, the compiler outputs an error message.

Arrays of pointers to various types of objects can be declared using complex declarators.

Storage

The storage associated with an array type is the storage required for all of its elements. The element of an array are stored in contiguous and increasing memory locations from the first element to the last. No blanks separate the array element in storage. Arrays are stored in row major order. For example the following array consists of two rows with three columns each :

```
int list [2][3] ;
```

The three columns of the first row are stored first, before the three columns of second row. This means that the last subscript varies most quickly.

Limitations

∗ The size of an array is restricted to 65535 bytes.

Example 4.23

```
int values [25] ;             /* declares an array variable named values with 25 elements
                                 each having type int */
long two_dim_array [2][10] ;  /* declares a two dimensional array of long type having 20
                                 elements.*/

struct tag
{
        int ivar ;
        long lvar ;
} array_of_structures [10] ;
                              /* Declares an array of structures having 10 elements. */

char *arr[25] ;               /* declares an array of 25 char pointers */
char *arr[0] ;                /* compiler issues error message */
```

4.5.6 Pointer Declarations

Syntax :

*type-specifier [memory_function_qualifier_list] * [modification-spec] declarator ;*

A pointer declaration names a pointer variable and specifies the type of the object to which the variable points. A variable declared as a pointer holds a memory address.

The type-specifier gives the type of the object, which can be any basic, structure or union type. Pointer variables can also point to functions, arrays and other pointers.

By making type-specifier **void,** programmer can delay specification of the type to which the pointer refers. Such an item is referred to as a pointer to void (void *). A variable declared as a pointer to **void** can be used to point to an object of any type. However, in order to perform operations on the pointer or on the object to which it points, the type to which it points must be explicitly specified for each operation. Such conversion can be accomplished with a type cast.

The modification-spec can be either **const** or **volatile**, or both. These specify, respectively, that the pointer will not be modified by the program itself (**const**), or that the pointer may legitimately be modified by some process beyond the control of the program (**volatile**).

Example 4.24

```
char * volatile * const buffer ;
/* 'buffer' is a location in ROM, whose content is constant. But the contents of the location
pointed to by 'buffer' may change beyond the control of program */
```

A const modification-spec also qualifies the pointer to be in Read Only Memory (ROM). Each level of indirection in a pointer declaration must be qualified as const if that indirection points to a location in Read Only Memory (ROM).

Example 4.25

```
int * const ptr ;           /* ptr is a location in ROM, whose content points to a RAM location */
int const * const iptr ;    /* iptr is a location in ROM, whose contents also point to a location in
                               ROM */
```

The declarator names the variable and can include a type modifier. For example, if declarator represents an array, the type of the pointer is modified to pointer to array.

Storage

The amount of storage required for an address depends on the memory model selected. If the pointer points to near memory, the size of the pointer is 2 bytes. If the pointer points to far memory, the size of the pointer is 3 bytes.

Example 4.26

```
char __near * string ;             /*  a pointer to near character named string */
long __far *arr_of_pntrs [10] ; /*  array of pointers to far long */
```

4.6 FUNCTION DECLARATIONS AND PROTOTYPES

Syntax :

[sc-specifier] [type-specifier] declarator([declarator] [[,declarator]...])

A function declaration also called a function prototype establishes the name and return type of a function and may specify the types, formal parameter names and number of arguments to the function. A function declaration does not define the function body. It simply makes information about the function known to the compiler. This information enables the compiler to check the types of the actual arguments in ensuing calls to the function.

If the expression that precedes the parenthesized argument list in a function call consists solely of an identifier, and if no declaration is visible for this identifier, the identifier is implicitly declared exactly as if, in the innermost block containing the function call. This implicit declaration is visible only in that particular block.

The sc-specifier represents a storage-class specifier; it can be either **extern** or **static**.

The type specifier gives the function return type and the declarator names the function. If type specifier is omitted, the function is assumed to return a value of type **int.**

The formal parameter is described in subsection 4.6.1. The final declaration-list represents further declaration on the same line. These may be other functions returning values of the same type as the first function or declarations of variables whose type is same as the first function's return type. Each such declaration must be separated from its predecessors and successors by a comma.

4.6.1 Formal Parameters

Formal parameters correspond to the actual parameters that can be passed to a function. In a function declaration, parameter declaration establishes the number and types of the actual arguments. They can also include identifiers of the formal parameters. These parameter declaration influence the argument checking done on function calls that appear before the compiler has processed the function definition.

A partial list of formal parameters may be declared using the above syntax. The formal parameter list must contain at least one declarator. Variable number of parameters may be indicated by ending the list with a comma followed by three periods (,...) referred to as the "ellipsis notation". A function is expected to have at least as many arguments as there are declarators or type specifiers preceding the last comma.

Example 4.27

```
int function1 (int number_of_items,...) ;
```

Structure or Union variables may also be passed as actual arguments to functions. The formal parameter list may also contain parameters of structure or union type.

Example 4.28

```
int function2 (struct a_tag arg, union v_tag value) ;
```

Identifiers used to name the formal parameters in the prototype declaration are descriptive only. They go out of scope at the end of the declaration. Therefore, they need not be identical to the identifiers used in the declaration portion of the function definition. Using the same names may enhance readability, but this use has no other significance.

4.6.2 Return Type

Functions can return values of any type except arrays and functions.

Example 4.29

```
struct tag
{
        int a ;
        long b ;
} input_structures[10] ;

struct tag
get_structure (int value)
{
    return (input_structures [value]) ;
}
```

4.6.3 List Of Formal Parameters

All elements of the formal-parameter-list argument appearing within the parentheses following the function declarator are optional.

Syntax :

[type-specifier] [declarator[[,...][,...]]]

If formal parameters are omitted in the function declaration, the parentheses should contain the keyword **void** to specify that no arguments will be passed to the function. If the parentheses are left empty, no information about whether arguments will be passed to the function is conveyed and no checking of argument types is performed.

A declaration in the formal parameter list can contain only the **auto** storage class specifier. If the type specifier is included, it can specify the type name for any basic type or pointer type. The declarator can be formed by combining a type specifier, plus the appropriate modifier with an identifier. Alternately an abstract declarator, that is a declarator without a specified identifier, can be used. At section 4.10 abstract declarators are explained.

Example 4.30

```
void function (void) ;           /*  declares a function with no return value and no arguments
                                 */
long fn (int, char) ;            /*  declares a function, which takes two arguments of int and
                                     char type and which returns a value long */
char *strtok(char s[],char c) ;  /*  declares a function which returns a pointer to character
                                     and takes two arguments char array and char */
struct inv_type *sfn () ;        /*  declares a function that returns a pointer to a structure of
                                     type inv_type and the number of arguments and argument
                                     types are undefined */
```

4.6.4 Function Qualifiers For Functions

CCU8 supports **__noreg** keyword that can qualify functions only. When a function is qualified with **__noreg,** CCU8 passes arguments by stack not by register. If **__noreg** function specifier is specified in function definition, CCU8 takes the arguments from the stack and not from the register.

4.7 STORAGE CLASS SPECIFIERS

The storage class of a variable determines whether the item has a global lifetime or local lifetime. An item with a global lifetime exists and has a value throughout the execution of the program.

All functions have global lifetimes.

Variables with local lifetime are allocated new storage each time execution control passes to the block in which they are defined. When execution control passes out of the block, the variable no longer has meaningful values.

CCU8 provides the following 5 storage class specifiers.

1. auto
2. static
3. extern
4. typedef
5. register

Items declared with **auto** storage class specifier have local lifetimes. Items declared with **static** or **extern** specifier have global lifetimes.

The **typedef** specifier does not reserve storage and is called a storage class specifier only for syntactic convenience. It is described in section 4.9.2.

The **register** storage class specifier causes the compiler to store the variable in a register, if possible. Register storage accelerates access time and reduces code size. Variables declared with **register** storage class have the same visibility as **auto** variables.

If registers are not available when the compiler encounters a register declaration, the variable is given **auto** storage class and treated accordingly. For variables declared as **register**, the address operator (unary &) cannot be applied.

Example 4.31

```
register int count ;
register index ;
```

The storage class specifiers have distinct meanings because storage class specifiers affect the visibility of functions and variables as well as their storage class. The term visibility refers to the portion of the source program in which the function or variable can be referenced by name. An item with a global lifetime exists throughout the execution of the source program, but it may not be visible in all parts of the program.

The placement of variable and function declarations within source files also affect storage class and visibility. Declarations outside all function definitions are said to appear at the external level; declarations within function definitions appear at the internal level.

The exact meaning of each storage class specifier depends on two factors :

∗ Whether the declaration appears at the external or internal level

∗ Whether the item being declared is variable or function.

The following subsections describes the meaning of storage class specifiers in each kind of declaration and explain the default behavior when the storage class specifier is omitted from a variable or function declaration.

4.7.1 Variable Declarations At The External Level

In variable declarations at the external level (that is, outside all functions), the **static** and **extern** storage class specifiers can be used or the storage class specifier can be omitted entirely. The storage class specifier **auto** cannot be used at the external level.

Variable declarations at the external level are either definitions of variables (defining declarations) or references to variables defined elsewhere (referencing declarations).

An external variable declaration that also initializes the variable is a defining declaration of the variable.

A definition at the external level can take several forms :

* A variable declared with the **static** storage class specifier is a definition of that variable. Both **const** and non-const **static** variable can be initialized with a constant-expression. For example **static int** x; and **const static int** y = 10; are considered definitions of variables ‘x’ and ‘y’.
* A variable that is explicitly initialized at the external level are definitions of that variable. CCU8 allows initialization of both **const** and non-const specified variables at the external level. For example, **const int** i = 10 and **int** y = 20 are the definitions of the variable ‘i’. and ‘y’ respectively.

Once a variable is defined at the external level, it is visible throughout the rest of the source file in which it appears. The variable is not visible prior to its definition in the same source file. Also, it is not visible in other source files of the program, unless a referencing declaration makes it visible, as described below.

A variable can be defined only once at the external level. If **static** storage class is used, another variable can be defined with the same name and **static** storage class in a different source file. Since each static definition is visible only within its own source file, no conflict occurs.

The **extern** storage class specifier declares a reference to a variable defined elsewhere. The **extern** declaration can be used to make a definition in another source file visible or to make variable visible before its definition in the same source file. The **extern** declaration makes a variable visible throughout the remainder of the source file in which the declaration occurs.

For an **extern** reference to be valid, the variable must be defined only once at the external level. The definition can be in any of the source files that form the program.

One special case is the omission of both the storage class specifier and the initializer from a variable declaration at the external level; for example, the declaration **int** a; is a valid external declaration. This declaration can have one of two different meanings depending on the context:

∗ If there is an external declaration of a variable with the same name elsewhere in the program, the current declaration is assumed to be a reference to the variable in the defining declaration as if the **extern** storage class specifier has been used in the declaration.

∗ If there is no external declaration of a variable elsewhere in the program, the declared variable is allocated storage at link time. This kind of variable is known as communal variable. If more than one such declaration appears in the same program but in different source files, storage is allocated for the largest size declared for the variable. For example if file1 contains the declaration **int** i; and file2 contains the declaration **long** i; and file1 and file2 form part of a same program, then storage space for a **long** value is allocated for ‘i’ at link time.

Example 4.32

```
/* FILE1 */
extern int global_variable ;     /* reference to global_variable defined below */

main ()
{
        global_variable = global_variable + 100 ;
        file1_function () ;
}

int global_variable ;            /* definition of global_variable */



file1_function ()
{
        file2_function () ;
        global_variable -= 100 ;
}

/* FILE2 */
extern int global_variable ;     /*reference to global_variable*/
static int file2variable ;       /*definition of file2variable, file2variable visible only in
                                   FILE2 */

file2_function ()
{
         global_variable += 10 ;
         return ;
}
```

4.7.2 Variable Declarations At The Internal Level

The storage class specifiers **auto**, **extern** and **static** can be used for variable declarations at internal level. When storage class specifier is omitted from such a declaration, the default storage class specifier is **auto**.

The local storage class specifier declares a variable with a local lifetime. An **auto** variable is visible only in the block in which it is declared. Declarations of **auto** variables can include initializer. Since **auto** variables are not initialized automatically, either they should be initialized explicitly or should be assigned initial values using statements within the block. The values of uninitialized auto variables are undefined.

A **static** local variable can be initialized with the address of any external or **static** item, but not with the address of a non static **auto** item, because the address of an auto item is not a constant.

A variable declared with the **static** storage class at the internal level has a global lifetime but is visible only within the block in which it is declared. Unlike **auto** variables, **static** variables keep their values upon exit from the block. A **const** qualified static variable is initialized only once, when the program execution begins; it is not initialized each time the block is entered.

A variable declared with the **extern** storage class specifier is a reference to a variable with the same name defined at the external level in any of the source files of the program. The internal **extern** declaration is used to make the external level variable definition visible within the block. Unless otherwise declared at the external level, a variable declared with the **'extern'** keyword at the internal level is visible only in the block in which it is declared.

Example 4.33

```
/********* FILE1 **********/
main ()
{
        extern int a ;          /* reference to 'a' defined in FILE2 */
          static int b ;        /* global lifetime, visible only within this function */
          int c = 0 ;           /* default storage class is auto, initialized to zero each time
                                   control enters this function */

        file2 () ;
}
```

```
/************FILE2 ************/
int a ;
int c ;

file2 ()
{
          int a ;                    /* Global 'a' is redefined, global 'a' is no longer visible */
          static int * d = &c ;      /* Address of global 'c' is used to initialize 'd'.
                                        Initialization is not done each time control enters the
                                        function, it is done only during the beginning of
                                        execution */
          a = c ;
}
```

4.7.3 Function Declarations At The Internal And External Levels

Function declarations can have either the **static** or **extern** storage class specifiers. Functions always have global lifetime.

The visibility rules for functions vary slightly from the rules for variables as follows :

* A function declared to be **static** is visible only within the source file in which it is defined. Functions in the same source file can call **static** functions, but functions in other source files cannot. Another **static** function with the same name in a different source file can be used without conflict.
* Functions declared as **extern** are visible throughout all the source files that make up the program, unless it is later redeclared as static. Any function can call an **extern** function.
* Function declarations that omit the storage class specifier are **extern** by default.

4.8 INITIALIZATION

Syntax :

= initializer

A variable can be set to an initial value by applying an initializer to the declarator in the variable declaration. The value or values of the initializer is assigned to the variables. An equal sign (=) precedes the initializer.

The following rules apply for initialization :

* Both const and non-const qualified variables declared at the external level can be initialized. If **const** qualified variables are not initialized at external level, they are assigned value 0.
* Variables declared with auto storage class specifier are initialized each time control passes to the block in which they are declared. If an initializer is omitted from the declaration of an auto variable, the initial value of the variable is undefined. Both aggregate (array, structure and unions) and non aggregate variables can be initialized.
* The initial values for external variable declaration and for all static variables, whether external or internal, must be constant expressions. Either constant or variable values can be used to initialize auto variables.
* The const qualifier also causes an item to be placed in Read Only Memory (ROM).

Example 4.34

```
char *volatile input_buf ;
const int integer_var1, integer_var3 ;
int integer_var2 ;
long long_var = 4 ;                               /* CORRECT */
const char * error_ptr = “pointer” ;              /* CORRECT*/
const char * const ptr = “pointer”;               /* CORRECT */
char * volatile * const buffer = &input_buf ;     /* CORRECT */
const int * var_ptr = &integer_var1 ;             /* CORRECT*/
int * const var_ptr1 = &integer_var2 ;            /* CORRECT */
const int * const var_ptr2 = &integer_var3 ;      /* CORRECT */
```

The following subsections describe how to initialize variables of fundamental, pointer and aggregate types.

4.8.1 Fundamental And Pointer Types

Syntax :

= expression

The value of expression is assigned to the variable. The conversion rules for assignment apply. Refer Sec 5.31.

An internally declared **static** variable can only be initialized with a constant value. Since the address of any externally declared or **static** variable is constant, it may be used to initialize an internally declared static pointer variable. However the address of an **auto** variable cannot be used as an initializer because it may be different for each execution of the block. However, if **/Za** option is specified, extern variables cannot be initialized.

Example 4.35

```
long lv = 100 ;                        /* lv is initialized to the constant value 100 */
static const int * const scp = 0 ;    /* The pointer scp is initialized to zero */
int x ;
int * const y = &x ;                   /* The pointer y is initialized with address of x */
int z = 10;                            /* data memory variable z is initialized to 10 */
const int m ;                          /* by default m is initialized to 0 */

func ()
{
        int local1 = 10 ;                  /* legal initialization */
        static int local = 100 ;
        int *p = &z ;                      /* valid, address of global variable can be used in
                                              initialization */
        static int *const lp = &local1 ;   /* invalid, address of local variables cannot be used to
                                              initialize a static variable */
}
```

4.8.2 Aggregate Types

Syntax :

= {initializer-list}

The initializer-list is a list of initializers separated by commas. Each initializer in the list is either a constant expression or an initializer list. Therefore, an initializer-list enclosed in braces can appear within another initializer-list. This form is useful for initializing aggregate members of aggregate type.

For each initializer-list, the values of the constant expressions are assigned, in order, to the corresponding members of the aggregate variable. When an union is initialized, initializer-value is assigned to the first member of the union.

If initializer-list has fewer values than an aggregate type, space is reserved for the remaining members or elements of the aggregate type. If initializer-list has more values than an aggregate type, an error results. These rules apply to each initializer-list, as well as to the aggregate as a whole.

Example 4.36

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

The above example declares and initializes ‘x’ as an one-dimensional array with three members, since no size is specified and there are three initializers.

Example 4.37

```
long y [4][3] = {
                        {1, 4, 7},
                        {2, 5, 8},
                        {3, 6, 9},
                  } ;
```

is a completely-bracketed initialization: 1,4 and 7 initialize the first row of the array y[0] namely y[0][0], y[0][1] and y[0][2]. Likewise the next two lines initialize y[1] and y[2]. The initializer ends early and, therefore, space is reserved for the elements of y[3]. Precisely the same effect could have been achieved by

Example 4.38

```
long y [4][3] = { 1,4,7,2,5,8,3,6,9 } ;
```

The initializer for ‘y’ begins with the left brace, but that for y[0] does not; therefore, three elements from the list are used. Likewise the next three are taken successively for y[1] and y[2].

The initialization

Example 4.39

```
const long y [4][3] = { {1}, {2}, {3}, {4} } ;
```

initializes the first column of the array, namely y[0][0], y[1][0], y[2][0] and y[3][0] with 1,2,3 and 4 respectively and reserves space for the rest. As the variable is qualified with **const**, the remaining locations are initialized to 0.

4.8.3 String Initializers

Syntax :

= “characters”

An array of characters can be initialized with a string literal. For example,

Example 4.40

```
char str_arr [] = “abc” ;
```

initializes str_arr as a four element array of characters. The fourth element is the null character which terminates all string literals. If array size is specified and the string is longer than the specified array size, the extra characters are simply ignored and a warning message is displayed. For example, the following declaration initializes str_arr as a three element character array.

Example 4.41

```
const char str_arr[3] = “abcd” ;
```

Only the first three characters of the string are assigned to ‘str_arr’. The character ‘d’ and the string terminating null character are discarded. This creates an unterminated string and a warning message is generated indicating the condition. If the string is shorter than the specified array size, space is kept aside for the remaining elements of the array.

4.9 TYPE DECLARATION

A structure or union type declaration defines the name and members of a structure or union type. The name of a declared type can be used in variable or function declarations to refer to that type. This is useful if many variables and functions have the same type.

A typedef declaration defines a type specifier for a type. A typedef declaration can be used to form shorter or more meaningful names for types already defined by or for types declared by the programmer.

4.9.1 Structure And Union Types

Declarations of structure and union types have the same general form as variable declarations of those types. However, structure and union type declarations and structure and union variable declarations differ in the following ways :

* In structure and union type declarations variable is omitted.
* In structure and union type declaration tag is required; it names the structure or union type.
* The member declaration list defining the type must appear in the structure and union type declaration.

Example 4.42

```
struct tag {
                int x ;
                char arr[20] ;
           } ;
```

The above example declares a structure type named tag.

4.9.2 Typedef Declarations

Syntax :

typedef type-specifier declarator[,declarator]...;

A **typedef** declaration is analogous to a variable declaration except that the '**typedef**' keyword replaces a storage class specifier. A **typedef** declaration is interpreted in the same way as a variable or function declaration, but the identifier, instead of assuming the type specified by the declaration, becomes a synonym for the type.

A **typedef** declaration does not create types. It creates synonyms for existing types, or names for types that could be specified in other ways. Any type including pointer, function and array types can be declared with **typedef**. A **typedef** name can be declared for a pointer to a structure or union type also.

Example 4.43

```
typedef int fixed_point ;      /*'fixed_point' is synonym for 'int'. Therefore declaring
                                 fixed_point x ; is equivalent to declaring int x ;*/

typedef struct {
            float y ;
            long x ;
        } COMPLEX ;

COMPLEX *sp ;
```

Declares COMPLEX as a structure type with 2 members. COMPLEX can be used in further declarations.

COMPLEX *sp ; /* declares a pointer sp to the structure of type COMPLEX */

4.10 TYPE NAMES

A type name specifies a particular data type, in addition to ordinary variable declarations and defined type declarations, type names are used in three contexts :

* In the formal parameter list of function declarations (prototypes)
* In type casts
* In sizeof operations.

Formal parameter lists are discussed in section 4.6.1.

The type names for fundamental, structure and union types are simply the type specifiers for those types. A type name for the pointer, array or function type has the following form:

type-specifier abstract-declarator

An abstract declarator is a declarator without an identifier, consisting of one or more pointer, array or function modifiers. The pointer modifier (*) always precedes the identifier in a declarator; array ([]) and function (()) modifiers always follow the identifier. Knowing this, one can determine where the identifier would appear and interpret the declarator accordingly.

Abstract declarators can be complex. Parentheses in a complex abstract declarator specify a particular interpretation, just as they do for the complex declarators in declarations. The type specifiers established by typedef declarations also qualify as type names.

Example 4.44

```
int * ;            /* type name for pointer to int */
long (*)[5] ;      /* type name for a pointer to an array of long elements */
int (*)(void) ;    /* typename for a pointer to a function, with no arguments and returning int
                      type. */
```

4.11 FUNCTIONS

A function is an independent collection of declarations and statements, usually designed to perform a specific task. ‘C’ programs have atleast one function, the ‘**main**’ function, and can have other functions. The following subsections describe how to define, declare and call ‘C’ functions.

4.11.1 Function Definitions

Syntax :

[sc-specifier][type-specifier] declarator([formal-parameter-list])
function-body

[sc-specifier][type-specifier] declarator([identifier-list])
[parameter-declarations]
function-body

[sc-specifier] [type-specifier] declarator([declarator] [,declarator]...])
function-body

A function definition specifies the name, formal parameters and body of a function. It also stipulates the return type and storage class of the function.

4.11.1.1 STORAGE CLASS

The sc-specifier in a function definition gives the function either **extern** or **static** storage class. If a function definition does not include a storage class specifier, the storage class specifier defaults to **extern.**

A function with static storage class is visible only in the source file in which it is defined. All other functions whether they are given **extern** storage class explicitly or implicitly, are visible throughout all the source files that make up the program.

If **static** storage class is desired, it must be declared on the first occurrence of the declaration (if any) of the function, and on the definition of the function.

4.11.1.2 RETURN TYPE AND FUNCTION NAME

The return type of a function establishes the size and type of the value returned by the function and corresponds to the type-specifier in the syntax of the function definition. The type can specify any basic type. If a type specifier is not included, the return type is assumed to be **int**.

The declarator is the function identifier, which can be modified to a pointer type. The parenthesis following the declarator establishes the item as a function.

The return type given in the function definition must match the return type in declarations of the function elsewhere in the program. Return type of a function is used only when the function returns a value. A function returns a value when a return statement containing an expression is executed. The expression is evaluated, converted to the function return value type, if necessary, and returned to the point at which the function was called. If no return statement is executed or if the return statement does not contain an expression, the return value is undefined.

If '**void**' keyword is used as a type specifier, then the function cannot return a value.

4.11.1.3 FORMAL PARAMETERS

Syntax :

form1:

[sc-specifier][type-specifier] declarator([formal-parameter-list])
function-body

form2:

[sc-specifier][type-specifier] declarator([identifier-list])
[parameter-declarations]
function-body

Formal parameters are variables that receive values passed to a function by a call. In form1 of syntax, the parentheses following the function name contain complete declarations of the formal parameters. The formal-parameter-list is a sequence of formal parameter declarations separated by commas.

In form2 of a function definition the formal parameters are declared following the closing parentheses, immediately before the beginning of the function body. In this form, the optional identifier-list is a list of identifiers that the function uses as the names of formal parameters. The order of the identifiers in the list determine the order in which they take on values in the function call. The identifier-list consists of zero or more identifiers, separated by commas. The list must be enclosed in parentheses, even if it is empty. The parameter-declaration establishes the type of the identifiers in form2.

If no arguments are to be passed, then the list of formal parameters can be replaced by the keyword ‘void’.

Formal parameter declarations specify the types, sizes and identifiers of values stored in the formal parameters. In form2 these have the same form as other variable declarations. In form1 each identifier in the formal-parameter-list must be preceded by its appropriate type specifier.

Example :4.45

```
/* function is defined in form1 */
void form1(long a, long b, long c)
{
        return ;
}

/* function is defined in form2 */
void form2 (a,b,c)
long b,c ;
long a ;
{
        return ;
}
```

The order and type of formal parameters must be same in all the function declarations, if any, and in the function definition. The types of the actual arguments in calls to a function must be assignment compatible with the types of the corresponding formal parameters. A formal parameter can have basic or pointer type.

The only storage class allowed for a formal parameter is **auto**. Undeclared identifiers in the parentheses following the function name have a default type **int**.

The identifiers of the formal parameters are used in the function body to refer to the value passed to the function. These identifiers cannot be redefined inside the function body, at the top level. However, they can be redefined in the inner blocks.

In form2 only identifiers appearing in the identifier list can be declared as formal parameters. In form2 the formal parameter declarations can be in any order.

The compiler, if necessary, performs the usual arithmetic conversion on each parameter. After conversion, no formal parameter is of type **char,** because all **char** declared formal parameters are converted to type **int.**

4.11.1.4 FUNCTION BODY

A function body is a compound statement containing statements that define what the function does. It may also contain declarations of variables used by these statements.

All variables declared in a function body have auto storage class unless otherwise specified. When the function is called, storage is created for the local variables. A return statement containing an expression must be executed inside the function body if the function is to return a value.

4.11.2 Function Prototypes

A function prototype declaration specifies the name, return type and storage class of a function. It can also establish types and identifiers of some or all of the arguments. The prototype has the same form as the function definition, except that it is terminated by a semicolon immediately following the closing parenthesis and therefore has no body.

If a call to a function precedes its declaration or definition a default prototype of the function is created by the compiler, giving it **"int**" return type. The types and the number of arguments used are the basis for declaring the formal parameters. Thus a call to a function is an implicit declaration, but the prototype generated may not adequately represent a subsequent call or definition of the function. This implicit declaration is valid only for the block containing the function call.

A prototype establishes the attributes of a function so that calls to the function that precede its definition can be checked for argument and return type mismatches. If the **static** storage class is specified in a prototype, then the static storage class must be specified in the function definition also.

Function prototypes have the following important uses :

* They establish the return types of functions that return a type other than int. If such a function is called before definition or declaration the results are undefined.
* If the prototype contains a full list of parameter types, argument types occurring in a function call or definition can be checked. The parameter list in prototype declaration is used for checking the correspondence of actual arguments in the function call with the formal parameters in the function definition.
* Prototypes are used to initialize pointers to functions before those functions are defined.

4.11.3 Function Calls

Syntax :

expression([expression-list])

A function call is an expression that passes control and actual arguments, if any, to a function. In function call, expression evaluates to a function address and expression-list is list of expressions separated by commas. The values of these latter expressions are the actual arguments passed to the function. If the function takes no arguments the expression-list must be empty.

When the function is executed :

1. The expression in the expression-list is evaluated and converted using the usual arithmetic conversions. If a function prototype is available, the results of these expressions may further be converted consistent with the formal parameter declarations.
2. The expression in expression-list are passed to the formal parameters of the called function. The first expression in the list always corresponds to the first formal parameter of the function, the second expression corresponds to the second formal parameter and so on through the list. Since the called function uses copies of the actual arguments, any changes it makes to the arguments do not affect the values of variables from which the copies may have been made.
3. Execution control passes to the first statement in the function.
4. The execution of a return statement in the body of the function returns control and possibly a value to the calling function. If no return statement is executed, control returns to the caller after the called function is executed. In such cases the return value is undefined.

4.11.3.1 ACTUAL ARGUMENTS

An actual argument can be any value with fundamental or pointer type. All actual arguments are passed by value. Pointers provide a way for a function to access a value by reference.

The expressions in a function call are evaluated and converted as follows :

* The usual arithmetic conversions are performed on actual argument in the function call. If a prototype is available, the resulting argument type is compared to the prototype’s corresponding formal parameter. If they don’t match, both conversion is performed and a diagnostic message is issued.
* If no prototype is available, default conversions are performed on each actual argument before it is passed to the function. In the default conversion, arguments of type ‘**char**’ are converted to type ‘**int’** and arguments of type ‘**float**’ are converted to type ‘**double**’. A prototype is created whose formal parameter types correspond to the types of the actual parameters after conversion.

The number of expressions in the expression-list must match the number of formal parameters in the function prototype or function definition. If the prototype formal parameter list contains only the ‘**void**’ keyword, the compiler expects zero arguments in the function call and the function definition. A diagnostic message is issued otherwise.

4.11.3.2 RECURSIVE CALLS

Any function in a ‘C’ program can be called recursively; that is, it can call itself. The ‘C’ compiler allows any number of recursive calls to itself. Each time the function is called, new storage is allocated for the formal parameters and for the auto variables, so that their values in previous, unfinished calls are not overwritten. Variables declared as **static** do not require new storage with each recursive call. Their storage exists for the lifetime of the program

4.12 ASM DECLARATION

The keyword **__asm** can be used to specify a ‘ASM’ statement in the following format.

__asm (string)

The above statement can occur both outside and inside a function. The processing of “**__asm**” statement inside and outside a function is similar. Refer Sec 6.8.

5. EXPRESSIONS AND OPERATORS

5.1 OPERATORS

An expression is any series of symbols used to produce a value. The simplest expressions are constants and variable names. Other expressions combine operators and subexpressions to produce values.

‘C’ operators can be used in conjunction with simple variable identifiers and constants to create complex expressions. The ‘C’ operators fall into the following categories :

∗ Unary operators, which take single operand.

∗ Binary operators, which take two operands and perform a variety of arithmetic and logical operations.

∗ Conditional operator (a ternary operator), which takes three operands and evaluates either the second or third expression, depending on the evaluation of the first expression.

∗ Assignment operators which assign a value to a variable, also converts the right-hand value to the type of the left-hand value, before the assignment takes place.

∗ Comma operator which guarantees left to right evaluation of comma-separated expressions. The result is the right most expression.

Unary operators appear before their operand and associate from right to left. Binary operators associate from left to right. ‘C’ has one ternary operator and it associates from right to left.

The precedence and associativity of ‘C’ operators affect the grouping and evaluation of operands in expressions. An operator precedence is meaningful only if other operators with higher or lower precedence are present. Expressions with higher precedence operators are evaluated first.

The following table summarizes the precedence and associativity of ‘C’ operators, listing them in order of precedence from highest to lowest. Where several operators appear together in a line, they have equal precedence and are evaluated according to their associativity.

Precedence and Associativity of ‘C’ Operators :

Operators	**Associativity**
() [] -> .	Left to right
- + ~ ! * & ++ -- sizeof casts	Right to left
* / %	Left to right
+ -	Left to right
<< >>	Left to right
< <= > >=	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?:	Right to left
= += -= *= /= %= &= ^= |= <<= >>=	Right to left
,	Left to right

An expression can contain several operators with equal precedence. When several such operators appear at the same level in an expression, evaluation proceeds according to the associativity of the operator, either from right to left, or from left to right.

5.2 LVALUES AND RVALUES

A variable identifier is one of the ‘C’ primary expressions. This type of expression yields a single value, the object of the variable. However, when using the variable identifier with other operators, the expression evaluates to the location of the variable in memory. The address of the variable is the lvalue. The object stored at the address is the rvalue. CCU8 uses rvalue and lvalue of variables in evaluation of an expression given below :

```
x = y ;
```

The contents of variable y are assigned to variable x. In other words, the expression on the right evaluates to the rvalue while the expression on the left evaluates to the lvalue of the expression in performance of assignment.

The following ‘C’ expressions may be lvalue expressions :

∗ Identifier of scalar variables
∗ References to scalar elements
∗ References to structure and union variables
∗ References to structure and union members, except for references to fields which are not lvalues.
∗ References to pointers (also called dereferenced pointers; an asterisk(*) followed by an address valued expression)
∗ Any of the above expressions enclosed in parentheses.

The above is expressed as the following syntax for lvalue :

```
lvalue :
        identifier
        expression[expression]
        expression.expression
        expression->expression
        *expression
        (lvalue)
```

All lvalue expressions represent a single location in memory.

5.3 CONVERSIONS

Some operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions.

5.3.1 Integral Promotion

One of the following may be used in an expression wherever an integer may be used:

1. a character
2. an integer or character bit-field
3. an object of enumeration type.

If an int can represent all values of the original type, the value is converted to an **int**, otherwise, it is converted to an **unsigned int**. These are called the integral promotions. All other arithmetic types are unchanged by the integral promotions.

5.3.2 Arithmetic Conversions

Many operators cause conversions and yield result types in a similar way. The effect is to bring operands into a common type, which is also the type of the result. This pattern is called the usual arithmetic conversions.

* First, if either operand is **long double**, the other is converted to **long double**.
* Otherwise, if either operand is **double**, the other is converted to **double**.
* Otherwise, if either operand is **float**, the other is converted to **float**.
* Otherwise, the integral promotions are performed on both operands; then, if either operand is unsigned **long int**, the other is converted to **unsigned long int**.
* Otherwise, if one operand is **long int** and the other is **unsigned int**, both are converted to **unsigned long int**.
* Otherwise, if one operand is **long int**, the other is converted to **long int**.
* Otherwise, if either operand is **unsigned int**, the other is converted to **unsigned int**.
* Otherwise, both operands have type **int**.

5.3.3 Pointer Conversions

When two pointers are operated upon, they are converted to same size. Pointer size depends upon the memory model and the memory model qualifier specified for the pointer. When a pointer is qualified with **__far**, the size of the pointer is 3 bytes and when a pointer is qualified with **__near**, the size of the pointer is 2 bytes.

Expressions may contain, both near pointers (2 bytes) and __far pointers (3 bytes). When two pointers of different size are operated upon, they are promoted to same size. The near pointer is converted to far pointer, with default segment address in the upper byte.

5.4 PRIMARY EXPRESSIONS AND OPERATORS

Simple expressions are called primary expressions. Primary expressions are identifiers, constants, strings or expressions in parentheses.

```
primary_expression :
        identifier
        constant
        string
        (expression)
```

5.4.1 Identifiers

Identifier names a variable or function. Variables is one of the basic data objects manipulated in a program. Declarations list the variables to be used. Declarations also specify the type of the variable.

An identifier can be qualified as far variable by specifying the keyword **__far**, immediately to it’s left. A far variable need not be allocated in the default segment. Therefore, segment switching is done before accessing far variables.

5.4.2 Constants

A constant operand has type and value of the constant value it represents. Its type depends on its form. Character constants has **int** type. Enumerator constants also have **int** type. In general, the type of constants may be **int**, **unsigned int**, **long**, **unsigned long**, **float** or **double**.

5.4.3 Strings

A string literal is a character or sequence of adjacent characters enclosed in double quotation marks. Two or more adjacent string literals separated only by white space are concatenated into a single string literal. After concatenation, a null byte **'\0'** is appended at the end, so that programs that scan the string can find its end.

String literal is stored as an array of elements with char type in code memory. Its type is originally "array of const char". This is usually modified as "pointer to const char" and the result is the pointer to the first character in the string. The storage class of string literal is static.

5.4.4 Parenthesized Expression

A parenthesized expression is a primary expression whose type and value are identical to those without parentheses. Mainly, parentheses is used to change the associativity and precedence of operators.

Example 5.1

```
(5 + 5) * 3
```

In the above example, the parentheses around 5 + 5 mean that the value of 5 + 5 is the left operand of the multiplication operator (*). The result of the above expression is 30. Without parentheses, 5 + 5 * 3 would evaluate to 20.

5.5. ARRAY REFERENCES

array_reference :
    expression1 [expression2]

Bracket operators ([and]) are used to refer to elements of arrays. One expression followed by another expression in square brackets denote a subscripted array reference.

One of the two expression must have type “pointer to T”, where T is some type, and the other must have integral type and the resultant type of the subscript expression is T.

The expression expression1[expression2] is identical to *((expression1) + (expression2)) by definition, since both cases represent the value at the address that is expression2 positions beyond expression1.

5.6 FUNCTION CALLS

function calls :
    expression ( )
    expression (argument_list)

A function is an expression followed by parentheses. The parentheses may contain a list of arguments separated by commas or may be empty. The syntax of argument list is as shown below :

argument_list :
    expression
    argument_list, expression

Function calls may or may not have declaration preceding it. If declaration is not specified previously, return type is assumed to be of ‘int’ type.

The expression in the function call must be of type “pointer to function returning T’, for some type T. The resultant type of the function call is T.

In preparing for function call, a copy is made for each argument and all argument-passing is strictly by value. The called function may change the values of its parameters. However, these changes will not affect the values of the arguments in the calling function.

The arguments undergo integral promotion before being sent.

Error message is displayed if the number of arguments in function call disagrees with the number of parameters in the definition of the function, unless the parameter list ends with the ellipsis notation (...). In the latter case, the number of arguments must equal or exceed the number of parameters; trailing arguments beyond the explicitly typed parameters suffer default argument promotion.

If no prototype is specified for a function and if its body is not defined, the above mentioned checks are not performed. If prototype is not specified, CCU8 assumes the prototype from the body definition, if specified. If the prototype and the body definition differs, body definition overwrites the prototype.

The order of evaluation of arguments is from left to right. Recursive calls to any function is permitted.

When a far pointer is passed as an argument to a function which actually takes a near pointer, the segment information of the argument pointer is lost. CCU8 issues error when a far pointer is passed as argument in place of near pointer. However, if a near pointer is passed as argument to a function which actually takes a far pointer, a warning message is issued. The near pointer is converted to far pointer with the default segment address in the upper byte of the converted pointer.

The following built-in functions are supported.

```
__EI        __DI
```

The prototypes of the above built-in functions are given below:

```
void __EI (void) ;
void __DI (void) ;
```

These built-in functions may be called as any other function is called.

5.7 STRUCTURE AND UNION REFERENCES

```
structure_reference :
        expression . identifier
        expression -> identifier
```

A member of a structure or a union may be referenced with either of the two operators : the period (.) or the right arrow (->).

5.7.1 Dot operator

An expression followed by a period followed by an identifier refers to a member of a structure or union. The first operand expression must be a structure or a union, and the identifier must name a member of the structure or union.

The resultant value is the named member of the structure or union, and the resultant type is the type of the member. The resultant expression is an lvalue if the type of the member is not an array type.

5.7.2 Arrow operator

An expression followed by an arrow followed by an identifier also refers to a member of a structure or union. The first operand expression must be a pointer to structure or union, and the identifier must name a member of the structure or union to which the pointer points.

The result refers to the named member of the structure or union to which the pointer points and resultant type is the type of the member.

Example 5.2

```
struct example {
                int member1 ;
                int member2 ;
                struct example * ptr_to_struct ;
              } s_variable, struct_array [10] ;

1.  s_variable.ptr_to_struct = &s_variable ;
2.  (s_variable.ptr_to_struct)->member1 = 25 ;
3.  struct_array [7].member2 = 100 ;
```

In the above example:

1. The address of s_variable structure is assigned to ptr_to_struct member of the structure.
2. The pointer expression s_variable.ptr_to_struct is used with pointer selection operator (->) to assign a value to member member1.
3. An individual structure member is selected from an array of structures.

5.8 POST INCREMENT

```
post_increment :
        expression ++
```

Post increment is performed when an expression is followed by the operator ++. The resultant value is the value of the operand. After the value is noted, the operand is incremented by one. Resultant type is the type of the operand. Result of the expression loses its lvalue.

The operand must be an integral, floating or pointer type and must be a modifiable (non-const) lvalue expression. An operand of integral or floating type is incremented by an integer value 1. The operand of the pointer type is incremented by the size of the object it addresses. An incremented pointer points to the next object.

Example 5.3

```
int a, b ;

a = b ++ ;
```

In the above example, the value of ‘b’ is assigned to ‘a’ first and then ‘b’ is incremented.

5.9 POST DECREMENT

```
post_decrement :
        expression --
```

Post decrement is performed when an expression is followed by the operator --. The resultant value is the value of the operand. After the value is noted, the operand is decremented by one. Resultant type is the type of the operand. Result of the expression loses its lvalue.

The operand must be an integral, floating or pointer type and must be a modifiable (non-const) lvalue expression. An operand of integral or floating type is decremented by an integer value 1. The operand of the pointer type is decremented by the size of the object it addresses. A decremented pointer points to the previous object.

Example 5.4

```
int a, b ;
a = b -- ;
```

The value of ‘b’ is assigned to ‘a’ first and then ‘b’ is decremented.

5.10 PRE INCREMENT

pre_increment:
    ++ expression

An expression preceded by a ++ operator is an unary expression. The operand is incremented (++) by 1. The value of the expression is the value of the operand after the increment. The operand must be an lvalue. Other rules are similar to that of post increment (refer to section 5.8 for further details).

Example 5.5

```
int a, b ;
a = ++ b ;
```

The value of ‘b’ is incremented before assignment. The incremented value is assigned to ‘a’.

5.11 PRE DECREMENT

pre_decrement:
    -- expression

An expression preceded by a -- operator is an unary expression. The operand is decremented (--) by 1. The value of the expression is the value of the operand after the decrement. The operand must be an lvalue. Other rules are similar to that of post decrement (refer to section 5.9 for further details).

Example 5.6

```
int a, b ;
a = -- b ;
```

The value of ‘b’ is decremented before assignment. The decremented value is assigned to ‘a’.

5.12 ADDRESS OPERATOR

```
address_operator :
        & expression
```

Address may be computed using the address operator &. The unary & operator takes the address of its operand. The operand may be any value that is a valid lvalue of an assignment operation. However, neither a bit-field nor an object declared as register is allowed.

A warning message is displayed if an array name is the operand of an address operator. Since array names are addresses, & operator is ignored. No warning is issued, if a function designator is the operand of an address operator. The '&' operator is ignored for function designators.

The result is a pointer to the lvalue operand. If the type of the operand is T, the type of the result is "pointer to T".

Example 5.7

```
int x, * p ;

p = &x ;
```

The address operator (&) takes the address of x and assigns to p.

When address is taken for a variable that resides in far segment, the address size is 3 bytes. The address contains the offset value in the lower two bytes, and segment address in the upper byte.

Example 5.8

```
int __far x ;
int __far * p ;

p = &x ;            /* size of the address of x is 3 bytes */
```

5.13 INDIRECTION OPERATOR

```
indirection_operator :
        * expression
```

The unary * operator denotes indirection and is used for dereferencing a pointer. The operand must be a pointer value.

The result of the operation is the value addressed by the operand; that is the value at the address specified by the operand. Resultant type is the type the operand addresses. If the type of the expression is “pointer to T”, the type of the result is T.

Result is an lvalue, if the operand is not an array type.

Example 5.9

```
int x, * p ;
x = * p ;
```

The indirection operator (*) is used to access the integer value at the address stored in p. The accessed value is assigned to the integer x.

5.14 UNARY PLUS OPERATOR

```
unary_plus_operator :
        + expression
```

The operand of the ‘+’ operator must have arithmetic type, and the result is the value of its operand. Integral promotions are performed. The type of the result is the type of the promoted operand.

5.15 UNARY MINUS OPERATOR

```
unary_minus_operator :
        - expression
```

Unary minus operator (-) produces the negative (two’s complement) of its operand. The operand of the ‘-’ operator must have arithmetic type, and the result is the negative of its operand. Integral promotions are performed. Negative zero is zero. The type of the result is the type of the promoted operand.

Example 5.10

```
int variable ;
variable = 999 ;
variable = - variable ;
```

The value of variable is negative of 999, that is -999.

5.16 ONE’S COMPLEMENT OPERATOR

```
bit_not_operator :
~ expression
```

The operator ~ produces the bitwise complement of its operand. The operand of the ~ operator must have integral type, and the result is the one’s complement of its operand. Integral promotions are performed. The type of the result is the type of the promoted operand.

Example 5.11

```
unsigned int a, x ;
a = 0xaaaa ;
x = ~a ;
```

The value assigned to x is the one’s complement of the unsigned value 0xaaaa, that is 0x5555.

5.17 LOGICAL NOT OPERATOR

```
logical_not_operator :
        ! expression
```

Logical comparison is performed when an expression is preceded by the operator !. The operand must have arithmetic type or be a pointer.

Resultant value is either 1 or 0. Result is 1 if the value of the operand compares equal to zero, and 0 if the value of the operand is not zero. The type of the result is **int**.

Example 5.12

```
int x, y ;

if (! ( x < y) )
    fn () ;
```

If x is greater than or equal to y, the result of the expression is 1 (true). If x is less than y, the result is 0(false).

5.18 SIZEOF OPERATOR

```
sizeof_operator :
        sizeof expression
        sizeof (typename)
```

The sizeof operator yields the number of bytes required to store an object of the type of its operand. The operand is either an expression, which is not evaluated, or a parenthesized type name.

When sizeof operator is applied to a char, the result is 1; when applied to an array, the result is the total number of bytes in the array. The size of an array of ‘n’ elements is ‘n’ times the size of one element.

When the sizeof operator is applied to any structure or union, the result is the number of bytes in the object including any padding used to align the members of the structure or union on memory boundaries.

The sizeof operator may not be applied to an operand of incomplete type.

The result is an unsigned integral constant. Resultant type is unsigned int.

Typename is syntactically a declaration for an object of that type omitting the name of the object.

Example 5.13

```
long array [10] ;

z = sizeof ( array ) ;
```

The value of z is 40.

Sizeof operator can also be applied to expressions. These expressions are not evaluated. The result is the size of the result of the expression.

Example 5.14

```
int a, x, y, z ;

x = 10 ;
y = 10 ;
z = sizeof ( x = (y *2) ) ;
a = x ;
```

In the above example, the expression ‘x = (y * 2)’ is not evaluated. Therefore, value ‘10’ is assigned to ‘a’ and not ‘20’.

Example 5.15

```
int i, j ;
int * dptr ;

fn ()
{
        i =  sizeof (dptr) ;
}
```

In the above example, the size of the pointer variable ‘dptr’ depends on the data memory model in which it is compiled. If the program is compiled in near memory model, then the value of ‘i’ is 2. If the program is compiled in far memory model, then the value of ‘i’ is 3.

5.19 CAST OPERATOR

```
cast_operator :
        (typename) expression
```

Cast operator consists of data type name in parentheses. A unary expression preceded by the parenthesized name of a type causes conversion of the value of the expression to the named type. Typenames are discussed in detail in section 4.10.

If the operand is a variable, its data type is converted to the named type; the content of the variable is not changed.

Result of cast expression is not an lvalue, if the size of the typename is greater than the size of the expression.

A near variable cannot be casted to a far variable. However, a pointer to near memory can be casted to a pointer to a far memory. When a pointer to far memory is assigned to a pointer to a near memory, CCU8 issues warning. However, if a pointer to near memory is casted to a pointer to far memory, pointer conversion is performed.

Example 5.16

```
int x, y ;
int * cvar ;
int __far * dvar ;

dvar = ( int __far *) cvar ;    /* cvar is casted as far pointer and assigned to dvar */
```

5.20 MULTIPLICATIVE OPERATORS

```
multiplicative_expression :
        expression * expression
        expression / expression
        expression % expression
```

The multiplicative operators are *, / and %. They group from left to right.

The operands of * and / must have arithmetic type; the operands of % must have integral type. The usual arithmetic conversions are performed on the operands, and predict the type of the result.

The binary * operator denotes multiplication.

The binary / operator yields the quotient, and the % operator the remainder, of the division of the first operand by the second operand. If the second operand is zero, the result is undefined. If CCU8 detects the second operand as zero, warning message is displayed.

Example 5.17

```
unsigned int x, y, i, n, j ;
y = x * i ;
n = i / j ;
n = i % j ;
```

5.21 ADDITIVE OPERATORS

```
additive_expression :
        expression + expression
        expression - expression
```

The additive operators + and - group left to right. If the operands have arithmetic type, the usual arithmetic conversions are performed.

The result of + operator is the sum of the operands. A pointer to an object and a value of any integral type may be added. CCU8 calculates the size of one object, multiplies this by the integer thus obtaining the offset value, and then adds the offset value to the address of the designated element. The result is a pointer of the same type as the original pointer, and points to another object, appropriately offset from the original object. Thus if P is a pointer to an object, the expression P + 1 is a pointer to the next object.

CCU8 issues an error if two pointers are added.

The result of the - operator is the difference of the operands. A value of any integral type may be subtracted from a pointer; in that case the same conversions and conditions as for addition apply.

If two pointers to objects of the same type are subtracted, the result is a signed integral value representing the displacement between the pointed-to objects. The resultant value is calculated by finding the difference between the two pointers and dividing the difference by the size of the object to which the pointers point.

When two pointers are subtracted, only the offset value of the pointers are subtracted. Pointers pointing to objects of different types are not allowed.

Example 5.18

```
int x, y, i, j, k, l ;
char *p1, *p2 ;
long array1 [20], array2 [20] ;

y = x + i ;
p1 = p2 + 2 ;
j = &array1[k] - &array1[l] ;
```

5.22 SHIFT OPERATORS

shift_expression :
    expression << expression
    expression >> expression

The shift operators << and >> group from left to right. For both operators each operand must be integral, and is subject to integral promotion.

The type of the result is that of the promoted left operand.

The result of e1 << e2 is the value of the expression e1 shifted to the left by e2 bits. CCU8 clears vacated bits.

The result of e1 >> e2 is the value of the expression e1 shifted to the right by e2 bits. CCU8 clears vacated bits if the left operand e1 is unsigned; otherwise, vacated bits are filled with a copy of e1's sign bit.

The result is undefined if right operand is negative or greater than or equal to the number of bits in the left expression type.

Example 5.19

```
unsigned int x, y, z ;

x = 0x00aa ;
y = 0x5500 ;
z = ( x << 8) + ( y>>8) ;
```

In the above example, ‘x’ is left shifted eight positions and ‘y’ is shifted right eight positions. The shifted values are added giving 0xaa55, and assigned to ‘z’.

5.23 RELATIONAL OPERATORS

```
relational_expression :
        expression < expression
        expression > expression
        expression <= expression
        expression >= expression
```

The relational operators are less than (<), greater than (>), less than or equal to (<=) and greater than or equal to (>=).

The usual arithmetic conversions are performed on arithmetic operands. The result is 0 if the relation is false and is 1 if the relation is true. The resultant type is int.

Pointers to objects of same type may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is defined only for parts of the same object. If two pointers point to the same simple object, they compare equal; if the pointers are to members of the same structure, pointers to objects declared later in the structure compare higher; if the pointers are to members of the same union, they compare equal; If the pointer refers to members of an array, the comparison is equal to comparison of the corresponding subscripts.

A pointer may be compared with a constant integral expression with value 0, or with a pointer to void. However, if **/Za** option is specified, such comparisons are not supported.

The operators group from left to right. a < b < c is parsed as (a < b) < c, and a < b evaluates to either 0 or 1.

Example 5.20

```
const static int x = 10, y = 10 ;
int z ;

z = x > y ;
```

Since x and y are equal, the value 0 is assigned to z.

Example 5.21

```
char array [10], *p ;
void * v_ptr ;

for (p = array ; p < &array[10] ; p ++ )
*p = ‘\0’ ;

if ( v_ptr > p )      /* under /Za option error is issued */
      array [2] = ‘\0’ ;
```

The above program initializes each element of array to a null character constant.

5.24 EQUALITY OPERATORS

```
equality_expression :
        expression == expression
        expression != expression
```

The == (equal to) and the != (not equal to) operators are analogous to the relational operators except for their lower precedence. Thus a<b == c<d is 1 whenever a<b and c<d have the same truth value.

The equality operators follow the same rules as the relational operators. If both operands are pointers, pointer conversion is performed.

Two useful functions are provided to illustrate the use of equality operators :

Example 5.22

```
strcmp ( char s[], char t[])
{
      int i = 0 ;

      while ( s[i] == t[i] )
            if ( s[i ++] == ‘\0’ )
                  return (0) ;

      return ( s[i] - t[i] ) ;
}
```

The above function uses the equality operator. The above function returns a negative value if ‘s’ is less than ‘t’, and zero if ‘s’ is equal to ‘t’, and a positive value if ‘s’ is greater than ‘t’.

Example 5.23

```
squeeze( char s[], int c )
{
        int i, j ;

        for (i=j=0; s[i] != ‘\0’; i++)
             if ( s[i] != c )
                    s [j++] = s [i] ;

        s [j] = ‘\0’ ;
}
```

The above program removes all the occurrences of the character ‘c’ from the string ‘s’.

5.25 BITWISE AND OPERATOR

bit_and_expression :
    expression & expression

Bitwise And operator (&) may be used only with integral operands. The usual arithmetic conversions are performed.

The result is the corresponding bitwise AND function of the operands. Bitwise AND operator compares each bit of its first operand with the corresponding bit of the second operand. If both bits are 1, the resultant bit is 1; Otherwise the resultant bit is 0.

5.26 BITWISE EXCLUSIVE OR OPERATOR

bit_xor_expression :
    expression ^ expression

Bitwise Exclusive Or operator (^) may be used only with integral operands. The usual arithmetic conversions are performed.

The result is the corresponding bitwise exclusive OR function of the operands. Bitwise exclusive OR operator compares each bit of its first operand with the corresponding bit of the second operand. If one bit is zero and the other bit is 1, the resultant bit is set to 1; Otherwise the resultant bit is set to 0.

5.27 BITWISE OR OPERATOR

bit_or_expression :
    expression **|** *expression*

Bitwise inclusive OR operator (|) may be used only with integral operands. The usual arithmetic conversions are performed.

The result is the corresponding bitwise OR function of the operands. Bitwise OR operator compares each bit of its first operand with the corresponding bit of the second operand. If either bit is 1, the resultant bit is 1; Otherwise the resultant bit is 0.

5.28 LOGICAL AND OPERATOR

logical_AND_expression :
    expression && expression

The && operator is used for logical AND operation. Operator && groups from left to right. The result of the expression is either 1 or 0. Resultant type is int.

The operands need not have the same type, but each must have arithmetic type or pointer type.

If CCU8 is able to make an evaluation by examining only the left operand, it does not evaluate the right operand.

For the expression e1 && e2, first operand e1 is evaluated, including all side effects; If it is equal to 0, the value of the expression is zero and the second expression e2 is not evaluated. If it is non-zero, e2 is evaluated, and if it is equal to zero, the result is zero, otherwise one.

5.29 LOGICAL OR OPERATOR

logical_OR_expression :
    expression **||** *expression*

The || operator is used for logical OR operation. Operator || groups from left to right. The result of the expression is either 1 or 0. Resultant type is int.

The operands need not have the same type, but each must have arithmetic type or pointer type.

If CCU8 is able to make an evaluation by examining only the left operand, it does not evaluate the right operand.

For the expression e1 || e2, first operand e1 is evaluated, including all side effects; If it is non-zero, the value of the expression is one and the second expression e2 is not evaluated. If e1 is equal to zero, e2 is evaluated, and if it is equal to zero, the result is zero, otherwise one.

Example 5.24

```
xor ( int a, int b )
{
        return ( (a || b) && ! (a && b)) ;
}
```

The XOR operation is carried out by the above function. The XOR function returns a true value (1) when only one operand is true (non-zero). The above function illustrates the use of both logical AND and OR.

5.30 CONDITIONAL EXPRESSION AND OPERATORS

conditional_expression :
    expression1 ? expression2 : expression3

‘C’ has one ternary operator; the conditional operator (?:).

The expression1 must be integral, floating or pointer type. It is evaluated in terms of its equivalence to 0. Evaluation proceeds as follows :

If expression1 does not evaluate to zero, expression2 is evaluated and the result of the expression is the value of expression2.

If expression1 evaluates to 0, expression3 is evaluated and the result of the expression is the value of expression3.

Either expression2 or expression3 is evaluated but not both. Operator ? : groups from right to left.

The type of the result of a conditional operation depends on the type of expression2 or expression3 as follows :

∗ If expression2 or expression3 has integral or floating type, the operator performs usual arithmetic conversions. The type of the result is the type of the operands after conversion.

∗ If both expression2 and expression3 have the same structure, union or pointer type, the type of the result is the same structure, union or pointer type.

∗ If both operands have void type, the result has type void.

∗ If either operand is a pointer to an object of any type, and the other operand is a pointer to void, the pointer to the object is converted to a pointer to void and the result is pointer to void.

∗ If either of expression2 or expression3 is a near pointer and the other is a far pointer, pointer conversion is performed.

∗ If either expression2 or expression3 is a pointer and the other operand is a constant expression with the value 0, the type of the result is pointer type.

Example 5.25

```
int i, j ;
j = ( i < 0 ) ? (-i) : (i) ;
```

The above example assigns absolute value of i to j. If i is less than 0, -i is assigned to j. If i is greater than or equal to 0, i is assigned to j.

5.31 ASSIGNMENT EXPRESSIONS AND OPERATORS

assignment_expression :
expression assign_op expression

There are several assignment operators and all group from left to right. Assignment operators are one of :

```
=   +=   -=   *=   /=   %=   >>=   <<=   &=   |=   ^=
```

All require an lvalue as left operand, and the lvalue must be modifiable. It must not be an array, and must not have an incomplete type, or a function. Also its type must not be qualified with const. The type of an assignment expression is that of its left operand, and the value is the value stored in the left operand after the assignment has taken place.

In the simple assignment with =, the value of the expression replaces that of the object referred to by the lvalue. One of the following must be true :

* Both operands have arithmetic type, in which case the right operand is converted to the type of the left by the assignment.
* One operand is a pointer and the other is a pointer to void.
* The left operand is a pointer and the right operand is constant expression with value 0.
* Both operands are pointers to functions or objects whose types are the same except for the possible absence of const or volatile in the right operand.
* If a pointer to far memory is assigned to a pointer to near memory, the segment information is lost. Further operations using the pointer may result in undefined behavior. CCU8 issues error message if a far pointer is assigned to a near pointer. However, if a near pointer is assigned to a far pointer there is no loss of segment information. Default segment address will be assigned to the upper byte of the near pointer. CCU8 issues warning message, when a near pointer is assigned to a far pointer.

An expression of the form e1 op= e2 is equivalent to e1 = e1 op e2.

Example 5.26

```
float y ;
int x ;
y = x ;
```

The value of x is converted to float and assigned to y ;

Example 5.27

```
# define MASK 0Xff00
unsigned int n ;
n &= MASK ;
```

In the above example a bitwise AND operation is performed on ‘n’ and ‘MASK’, and the result is assigned to ‘n’.

5.32 COMMA EXPRESSION AND OPERATOR

comma_expression :
    expression, expression

The comma operator (,) evaluates its two operands sequentially from left to right. The result of the operation has the same value and type as the right operand. Each operand can be of any type. The comma operator does not perform type conversions between its operands.

The comma operator is typically used to evaluate two or more expressions in contexts where only one expression is allowed.

Example 5.28

```
1.  f (a, (t =3, t +2), c) ;
2.  for (i = 0,j = 0; i< 10; i ++, j +=2)
         array[i] = j ;       /* Array of Even nos */
```

The value of the second argument in the above example (1) is 5.

If the result of the comma operation is an array, then it is converted to pointer.

5.33 CONSTANT EXPRESSIONS

A constant expression is any expression that evaluates to a constant. The operands of a constant expression can be integral constants, character constants, floating-point constants, type casts, sizeof expressions and other constant expressions. Operators can be used to modify and combine operators.

Constant expressions used in preprocessor directives are subjected to certain restrictions. They cannot contain sizeof expressions, type casts to any type or floating-point type constants.

Constant expressions involving floating-point constants, cast to non-arithmetic types and address of expressions can only appear in initializers. The unary address-of operator (&) can be applied to variables with fundamental types that are declared at the external level or to subscripted array references.

6. STATEMENTS

6.1 INTRODUCTION

This section describes statements in ‘C’ language. Statements are executed in the order in which they appear, except where a statement explicitly transfers control to another location.

Statements are executed for their effect, and do not have values. They fall into several groups.

```
statements :
        labeled_statement
        expression_statement
        compound_statement
        selection_statement
        iteration_statement
        jump_statement
        asm_statement
```

Limits : The maximum number of levels to which compound statements, conditional statements and looping statements may be nested is restricted to 32.

6.2 LABELED STATEMENT

```
labeled_statement :
        identifier : statement
        case constant_exp : statement
        default : statement
```

Statements may carry label prefixes. A label consisting of an identifier declares the identifier. The only use of an identifier label is as a target to goto statement.

The scope of an identifier is the current function. Labels cannot be redeclared within the same function. Label names do not collide with identifiers with same name in other declarations (local as well as global). Because CCU8 uses a separate name space for labels.

A label, consisting of the keyword **case** followed by a constant expression, is a case label. A label consisting of the keyword **default** is called a default label. Case labels and default labels are used within the **switch** statements. If used elsewhere, error is displayed by CCU8. The constant expression of the case label must be of integral type. Case labels and default labels are explained in detail in the section for **switch** statement.

Labels in themselves do not alter the flow of control.

6.3 EXPRESSION STATEMENT

expression_statement :
  expression ;
  ;

Any valid expression can be used as a statement by terminating it with a semicolon. ‘C’ expressions are explained in section 5.

Most expression statements are assignments or function calls. All side effects from the expression are completed before the next statement is executed.

If the expression is missing, the construction is called a null statement; Null statements are used to provide null operations in situations where the grammar of the language requires a statement, but the program requires no work to be done.

Statements such as **do**, **for**, **if**, **while** require that an executable statement appear as the statement body. The null statement satisfies the syntax requirement in cases that do not need a statement body.

The following are examples of expression statements :

Example 6.1

```
int x, y, z, i ;

x = y + z ;         /* x is assigned the value of y + z */
i ++ ;              /* i is incremented */
```

Example 6.2

```
int i, table [100] ;

for (i = 0 ; i < 100; table[i++] = 0)
        ;
```

In this example, the loop expression of the for statement table[i++] = 0 initializes the first 100 elements of the array table to 0. The statement body is a null statement, since no further statements are necessary.

6.4 COMPOUND STATEMENT

```
compound_statement :
        { declaration_list statement_list }
        { declaration_list }
        { statement_list }
        { }
declaration_list :
        declaration
        declaration_list declaration
statement_list :
        statement
        statement_list statement
```

A compound statement is also called a block. Compound statements are provided so that several statements may be used, where a single statement is required by the language.

The compound statement contains optional declarations followed by a list of statements which is also optional, all enclosed in braces. If declarations are included, the variables declared are local to the block, and, for the rest of the block, they supersede any declarations of the variables of the same name. The outer declaration becomes valid at the end of the block.

Initializations of automatic objects included in the block are performed each time the block is entered in the order of the declarators. Initializations of static objects inside the block are performed only once.

Example 6.3

```
int y, z ;

fn ()
{
    int x = 10 ;

    z = 1 ;

    if (x > y)
        x++ ;
    else
        y++ ;
}
```

6.5 SELECTION STATEMENTS

Selection statements test the specified condition and depending on the result, one of several flows of control is chosen. There are two selection statements.

1. if statement
2. switch statement

6.5.1 if Statement

```
selection_statement :
        if (expression) statement
        if (expression) statement1 else statement2
```

‘if’ statements may or may not have the ‘**else’** part. The ‘**if**’ statement must have an expression in parentheses following the keyword ‘**if**’. Expression must be of arithmetic or pointer type. The expression is evaluated with all side effects.

If result of the expression is non-zero, then statement1 is executed. If the expression is zero, statement1 is not executed and statement2 is executed, if present.

When ‘**if**’ statements are nested within ‘**else**’ clauses, an ‘**else**’ clause matches the most recent ‘**if**’ statement that does not have an ‘**else**’ clause.

Example 6.4

```
int i, j, x ;

if (i < j)
        function (i) ;
else
{
        i = x ++ ;
        function (i) ;
}
```

6.5.2 switch Statement

```
selection_statement :
        switch (expression) statement
```

The '**switch'** statement transfers control to a statement within its body. Control passes to the statement whose **case** constant-expression matches the value of the **switch** expression. The **switch** statement may include any number of **case** instances. Execution of the statement body begins at the selected statement and proceeds until the end of the body or until a **'break'** statement.

The **'default'** statement is executed if no **case** constant expression is equal to the value of **switch** expression. If the **'default'** statement is omitted, and no **case** match is found, none of the statements in the **switch** body is executed. The **'default'** statement need not come at the end; it can appear anywhere in the body of the **'switch'** statement.

The type of the **switch** expression is integral. Each **case** constant expression is converted using the usual arithmetic conversions (explained in section 5.3.2). The value of each **case** constant expression must be unique within the statement body.

The **case** and **default** labels of the **switch** statement body are significant only in the initial test that determines where execution starts in the statement body. All statements, between the statement where execution starts and the end of the body, are executed regardless of their labels unless a statement transfers control out of the body entirely.

The following example illustrates the use of switch to display three different LED display items :

Example 6.5

```
display_fn ()
{
/* This program displays three types of displays based on the input */

int display_item ;

    while ( ( display_item = get_display_item () ) )
    {
        switch ( display_item )
        {
            case DATE : display_date () ;
            break ;

            case DAY : display_day () ;
            break ;
            case TIME : display_time () ;
            break ;
        }
    }
}
```

Declarations Within A Switch

Declarations may appear at the head of the compound statement forming the switch body. But initializations included in these declarations are not performed. The switch statement transfers control directly to an executable statement within the body, bypassing the lines that contain initializations.

Example 6.6

```
int y ;
switch (character)
{
        int x = 1 ;    /* Improper initialization */
        case ‘a’ :
                {
                        int x = 10 ;       /* Proper initialization */
                        y = x ;
                        break ;
                }
        case ‘b’ :
        .
        .
}
```

6.6 ITERATION STATEMENTS

Statements in the following subsections execute repeatedly (loop), until an expression evaluates to false.

6.6.1 for Statement

iteration_statement :
    for ([expression1] ; [expression2] ; [expression3]) statement

The **‘for’** statement evaluates three expressions and executes a statement (loop body) until expression2 evaluates to false. The ‘**for**’ statement is particularly useful for executing a loop body a specified number of times.

The ‘**for**’ statement executes the loop body zero or more times. It uses three optional control expressions as shown. A ‘**for**’ statement executes the following steps :

1. The optional expression1 is evaluated only once before the iteration of the loop. It usually specifies the initial values for variables.
2. The optional expression2 is evaluated before each iteration. If the expression evaluates to false, execution of the ‘**for**’ loop body terminates. If the expression is evaluated to true, the body of the loop is executed.
3. The optional expression3 is evaluated after each iteration. It usually specifies step value for variables initialized by expression1.
4. Iterations of the **‘for’** statement continue until expression2 produces a false value, or until some statement such as **break** or **goto** or **return** interrupts.

Example 6.7

```
int i ;
char string1 [20], string2 [20] ;

for (i = 0; i < 15; i++)
        string1 [i] = string2 [i] ;
```

The above example copies the first 15 characters of string2 to string1.

The following ‘for’ statement illustrates an infinite loop :

Example 6.8

```
int i, j ;

for ( ;; )
{
            j = i + 10 ;
}
```

Infinite loops can be terminated with a **goto**, **break** or **return** statement.

6.6.2 while Statement

iteration_statement :
while (expression) statement

The '**while**' statement evaluates an expression and executes a statement (loop body) zero or more times, until the expression evaluates to false.

If the expression in parentheses evaluates to false at the first time, the loop body never executes.

Example 6.9

```
int x, array [15] ;

fn ()
{
    x = 0 ;

    while (x < 10)
    {
        array [x] = x ;
        x++ ;
    }
}
```

The above example assigns the values 0 to 9 to the first ten elements of array.

6.6.3 do Statement

iteration_statement :
do statement while (expression) ;

The '**do**' statement executes a statement (the loop body) one or more times until the expression in the while clause evaluates to false.

The statement is executed at least once, and the expression is evaluated after each subsequent execution of the loop body. If the expression is true the statement is executed again.

Example 6.10

```
int num ;

do
{
    num = get_number () ;
} while (num <= 100) ;
```

The above example gets a number until it is greater than 100.

6.7 JUMP STATEMENTS

Jump statements transfer control unconditionally. The following statements are classified as jump statements.

1. goto statement
2. break statement
3. continue statement
4. return statement

Statements other than the '**goto**' statement may be used to interrupt the execution of another statement. These statements are primarily used to interrupt 'switch' statements and loops.

6.7.1 goto Statement

```
jump_statement :
        goto identifier ;
```

The '**goto**' statement transfers control automatically to a labeled statement, where the label identifier must be located in the scope of the function containing the goto statement.

Like other 'C' statements, any of the statements in a compound statement can carry a label. A **goto** statement can transfer into a compound statement. However, transferring into a compound statement is dangerous when the compound statement includes declaration that initialize variables. Since declarations appear before the executable statements in a compound statement, transferring directly to an executable statement within the compound statement bypasses the initialization. The results are undefined.

The following example illustrates both the '**goto**' statement and the labeled statement :

Example 6.11

```
int error_no ;

fn ()
{
    extern int error ;
    if (error )
        goto error_process ;

    .
    error_process :
    return (error_no) ;
}
```

In the above example, '**goto**' statement transfers control to the point labeled error_process.

6.7.2 break Statement

jump_statement :
    break ;

The **break** statement terminates the immediately enclosing **while**, **do**, **for** or **switch** statement. Control passes to the statement following the loop body.

Example 6.12

```
int x ;

while (1)
{
    x = fn () ;

    if (x == 1)
        break ;
}
```

In this example the while loop is executed until the function returns a value 1.

6.7.3 continue Statement

jump_statement :
    continue ;

The **continue** statement passes control to the end of the immediately enclosing **while**, **do** or **for** statement. The control passes to the next iteration of the **while**, **do** or **for** statement in which it appears, bypassing any remaining statements in the loop body.

Example 6.13

```
even_fn ()
{
    int x ;

    for (x = 0; x <= 100; x++)
    {
        if (x%2)
            continue ;

        print (x) ;
    }
}
```

The above function prints all the even numbers between 0 and 100.

6.7.4 return Statement

jump_statement :
    return [expression] ;

The return statement causes a return from a function with or without a return value.

CCU8 evaluates the expression, if one is specified, and returns the value to the calling function. If necessary, compiler converts the value to the declared type of the function. If there is no specified return value, the value is undefined.

Example 6.14

```
max (int a, int b)
{
    if (a > b)
        return (a) ;
    else
        return (b) ;
}
```

The above function returns the larger of its two integer arguments.

When a function which is declared to return nothing (void) returns a value, compiler issues a error message.

6.8 ASM STATEMENTS

asm_statement :
    __asm (“string”)

The asm_statement can be used to output the string contents in the assembly output file directly. The string is not processed by the compiler.

Example 6.15

INPUT:

```
        __asm (“; Test for __asm”)
        int gvar ;

        fn (int arg)
        {
                gvar = arg ;
                __asm (“\tl        er0,      NEAR _gvar\n”) ;

        }
```

OUTPUT :

```
CFILE 0000H 0000000AH "test.c"
; Test for __asm
        rseg $$NCODtest
_fn     :
;;gvar = arg ;
        st      er0,    NEAR _gvar
;;__asm ("\tl   er0,    NEAR _gvar\n") ;
        l       er0,    NEAR _gvar
;;}
        rt
        public _fn
        _gvar comm data 02h #00h
        extrn code near : _main

        end
```

7. VARIATIONS FROM ANSI STANDARD

The implementation of C language in CCU8 differs from the standard ANSI X3. 159-1989 and ISO/IEC 9899 proposed by ANSI (American National Standards Institute) due to the following features :

1. Supports specification of INTERRUPT functions.
2. Members of a structure or union cannot be qualified by '**const**'.
3. **Char** bit fields are allowed in structures and unions
4. Arguments cannot be qualified by '**const**'.
5. A declaration without a type specifier, a type qualifier or a storage class specifier is considered as declaration of type '**int**'.
6. A pointer can be compared to a constant integral expression, or to a pointer to void using relational operators also.
7. **__near** and **__far** memory model qualifiers are supported.
8. **__noreg** function qualifiers is supported.
9. **__asm** keyword is supported.
10. Single line comment (//) is supported.
11. The following built in functions are supported

    __EI **__DI** **__segbase_n** **__segbase_f** **__segsize**